



CentraleSupélec

2ème Année

CENTRALESUPÉLÉC METZ

Rapport de projet long

Pilotage semi-autonome de drone en espace clos à l'aide de Machine Learning



Elèves :

ADRIEN BARDOU

JAD ABDUL RAHMAN OBEID

MATHILDE DESAUTEL

Professeurs encadrants :

HERVÉ FREZAT-BUET

JÉREMY FIX

Année scolaire 2018-2019

Table des matières

1	Introduction	4
2	Environnement de travail et logiciels utilisés	6
2.1	Système d'exploitation	6
2.2	Logiciels utilisés	6
2.2.1	ROS	6
2.2.2	Unity	6
2.2.3	Langage de développement et bibliothèque	7
3	Traitement d'image	8
3.1	Traitement d'image pour la partie Navigation dans un couloir	8
3.1.1	Introduction	8
3.1.1.1	L'algorithme en une seule image	8
3.1.2	Détection des lignes	9
3.1.2.1	Utilisation de "Hough Line Transformation"	9
3.1.2.2	Utilisation de Line Segment Detector(LSD)	9
3.1.3	Détection de point de fuite	10
3.1.3.1	Définition	10
3.1.3.2	Représentation des lignes	11
3.1.3.3	Localisation du point de fuite	12
3.1.3.4	Clustering	13
3.1.3.5	Filtrage du flux des valeurs	14
3.1.3.6	La différence entre les pentes	15
3.1.4	Résultats	15
3.1.4.1	Simulation	15
3.1.4.2	Vrai Drone	17
3.2	Traitement d'image de la partie Escalier du projet	18
3.2.1	Introduction	18
3.2.2	Preprocessing	20
3.2.2.1	Détection des lignes avec LSD	20
3.2.2.2	Extraction des segments intéressants	21
3.2.3	Extraction des segments de l'escalier	22
3.2.3.1	Explication du clustering	23
3.2.3.2	Formation des clusters	24
3.2.3.3	Affichage des clusters	25
3.2.3.4	Détection des segments de l'escalier	26
3.2.3.5	Isolement de l'escalier	27
3.2.4	Extraction de l'information de l'escalier	28
3.2.4.1	Comment aller vers l'escalier ?	28
3.2.4.2	Méthode du trapèze	31
3.3	Traitement d'image de la partie Détection des portes ouvertes	36
3.3.1	Introduction	36
3.3.2	Flux optique en quelques mots	36
3.3.3	L'algorithme de Lucas-Kanade	36

3.3.3.1	Idée	36
3.3.3.2	Problème	36
3.3.3.3	Solution	37
3.3.3.4	Problème	37
3.3.4	Flux optique dense	38
3.3.5	Estimation de la distance	38
3.3.5.1	Test dans l'Ecole	40
3.3.6	Décision	40
3.3.7	L'algorithme en une seul image	41
4	Commande du drone avec ROS	42
4.1	Les modules pour la commande avec vitesse constante	42
4.1.1	Résultats :	42
4.2	Commande pour tourner d'un angle donné	44
4.2.1	Les Quaternions	45
4.2.2	Calcule de commande	45
4.3	Commande pour naviguer dans les couloirs	46
4.3.1	Introduction	46
4.3.2	Aller droit	46
4.3.3	Se déplacer au centre du couloir	47
4.4	Commande pour le passage d'escaliers	49
4.4.1	Introduction	49
4.4.2	Phase de placement	49
4.4.2.1	Stratégie	50
4.4.2.2	Localisation de l'escalier	51
4.4.2.3	Élaboration de la commande	52
4.4.2.4	Fin de tâche	54
4.4.2.5	Problèmes rencontrés et solutions	54
4.4.3	Phase de montée	54
4.4.3.1	Stratégie générale	54
4.4.3.2	Élaboration de la commande	56
4.4.3.3	Fin de tâche	57
4.4.3.4	Problèmes rencontrés et solutions	57
4.5	Commande pour la recherche des portes ouvertes	61
4.5.1	Introduction	61
4.5.1.1	Étape 1	61
4.5.1.2	Étape 2	61
5	Le Séquenceur	62
5.1	Introduction	62
5.2	Structure du projet	62
5.3	Le Grafcet	62
5.4	Convention de nomenclature	63
5.4.1	Le séquenceur	63
5.4.2	La séquence	63
5.4.3	La phase	64
5.4.4	Le projet en grafcet	65

5.4.5	La télécommande	65
5.4.6	Le graphe du projet	67
6	Appendice	68
6.1	Appendice A	68
6.1.1	Simulations	68
6.1.1.1	Introduction	68
6.1.1.2	Lancement ROSCORE	68
6.1.1.3	Lancement ROSBRIDGE	68
6.1.1.4	Éditeur Unity3D	68
6.1.1.5	ROSLaunch pour votre projet	68
6.1.1.6	Résultats	69
6.2	Appendice B	70
6.2.1	Connection au vrai drone	70
6.2.1.1	Introduction	70
6.2.1.2	Getting Ready	70
6.2.1.3	Étapes	70
6.2.1.4	Le décollage et l'atterrissement	72
6.3	Appendice C	73
6.3.1	Filmer avec ROSBAG	73
6.3.1.1	Qu'est-ce que RosBag ?	73
6.3.1.2	Comment utiliser RosBag	73
6.3.1.3	Enregistrez tous les topics :	73
6.3.1.4	Jouer un fichier bag spécifique :	73
6.3.2	Enregistrement d'une Vidéo	73

1 Introduction

Les progrès technologiques ont rendu réel ce qui étaient encore utopiques il y a plusieurs années. Les drones sont un des fruits de ce développement. Leur utilité n'est plus à démontrer, leur potentiel est vaste et leur utilisation est variée. Des drones sauveteurs aux drones policiers, en passant par ceux dont l'utilisation n'est autre que ludique, ils sont de plus en plus répandus, leur prix est de plus en plus attractif et leur qualité augmente. On peut compter parmi les drones de divertissement le drone Bebop 1 commercialisé par Parrot. Il permet de prendre des photos haute définition, son pilotage est aisé et son prix abordable.

Ce drone est stable en plein air. Cependant, il devient instable dans les espaces clos (tels que les couloirs) du fait de la perturbation due à l'air brassé par ses hélices. Son utilisation est donc difficile à l'intérieur d'un bâtiment mais pas impossible. Le but de ce projet est d'aider à son pilotage en espace clos.

Un des points d'attractivité du campus de Metz de l'école CentraleSupélec est son pôle dédié au pilotage des drones. Cependant pour que cela soit réalisable, il faut que ceux-ci soient stables. Pour atteindre cet objectif, nous allons aider à sa manipulation en créant un module d'assistance au pilotage, permettant de faciliter son utilisation en intérieur, notamment dans les couloirs et dans les escaliers où leur manipulation est délicate.

Ce projet sera scindé en trois parties :

- La première est la stabilisation des drones dans les couloirs, en utilisant le point de fuite créé par les lignes du couloir, partie réalisée par Jad Obeid.
- La seconde sera de mettre en place une détection d'escaliers afin d'automatiser leur montée, partie du projet réalisée par Mathilde Desautel et Adrien Bardou.
- La troisième partie sera la détection d'une porte ouverte orthogonale au couloir, partie réalisée par Jad Obeid.

Afin de réaliser cela, nous avons décidé d'utiliser une configuration souple qui est la suivante. On envoie un mode à un séquenceur. Ce séquenceur se chargera d'activer les noeuds nécessaires au traitement et calculs des données.

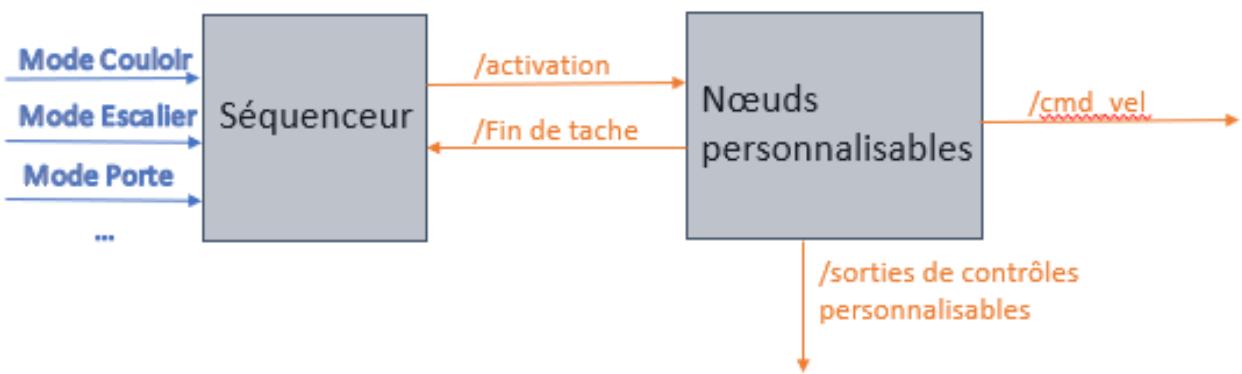


Figure 1 – Schéma global numéro 1

Afin de réaliser cela, nous avons suivi le plan suivant. La première phase est le traitement de l'image

reçue par le drone, que nous avons développé pour chacun des modes. La seconde est la commande du drone à l'aide de ROS : d'une part, élaboration de la commande en vitesse puis conversion en commande compréhensible par le drone. La troisième est la mise en place d'un séquenceur qui gérera l'activation des noeuds adéquats en fonction du mode sélectionné par l'utilisateur.

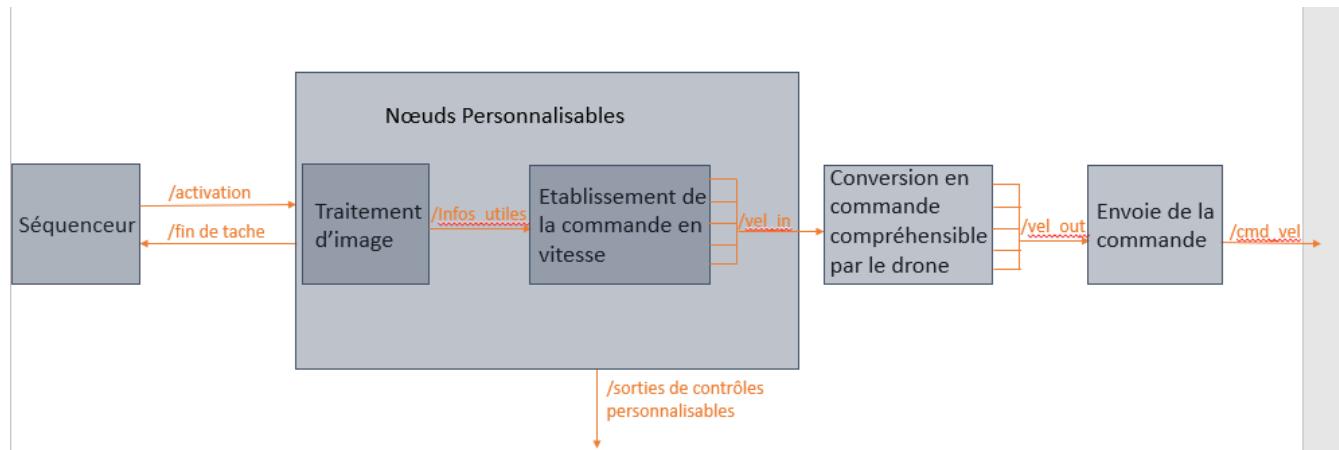


Figure 2 – Schéma global numéro 2

2 Environnement de travail et logiciels utilisés

2.1 Système d'exploitation

Ubuntu est un système d'exploitation open source développé par la société Canonical sur la base de la distribution Linux Debian. Dans notre projet, nous utilisons ROS comme environnement de développement pour le contrôle du drone. Nous avons choisi Ubuntu 18.04 pour développer notre projet. En effet, ROS ne possède pas un package .deb sur Ubuntu 18.10, il faut donc pour l'instant faire un "build from source".

2.2 Logiciels utilisés

2.2.1 ROS



Figure 3 – Logo ROS

Nous avons utilisé le framework ROS (Robot Operating System) qui est un framework open source permettant de développer des applications pour robot. Nous avons utilisé la version ROS melodic. D'après ROS , "ROS (Robot Operating System) fournit des bibliothèques et des outils pour aider les développeurs de logiciels à créer des applications robotiques. Il fournit une abstraction matérielle, des pilotes de périphériques, les bibliothèques, les outils de visualisation, le passage de messages, la gestion des paquets, et plus encore. ROS est sous licence open source, licence BSD."

2.2.2 Unity



Figure 4 – Logo Unity

Nous avons utilisé le logiciel Unity afin de créer des environnements de travail virtuels pour tester nos programmes.

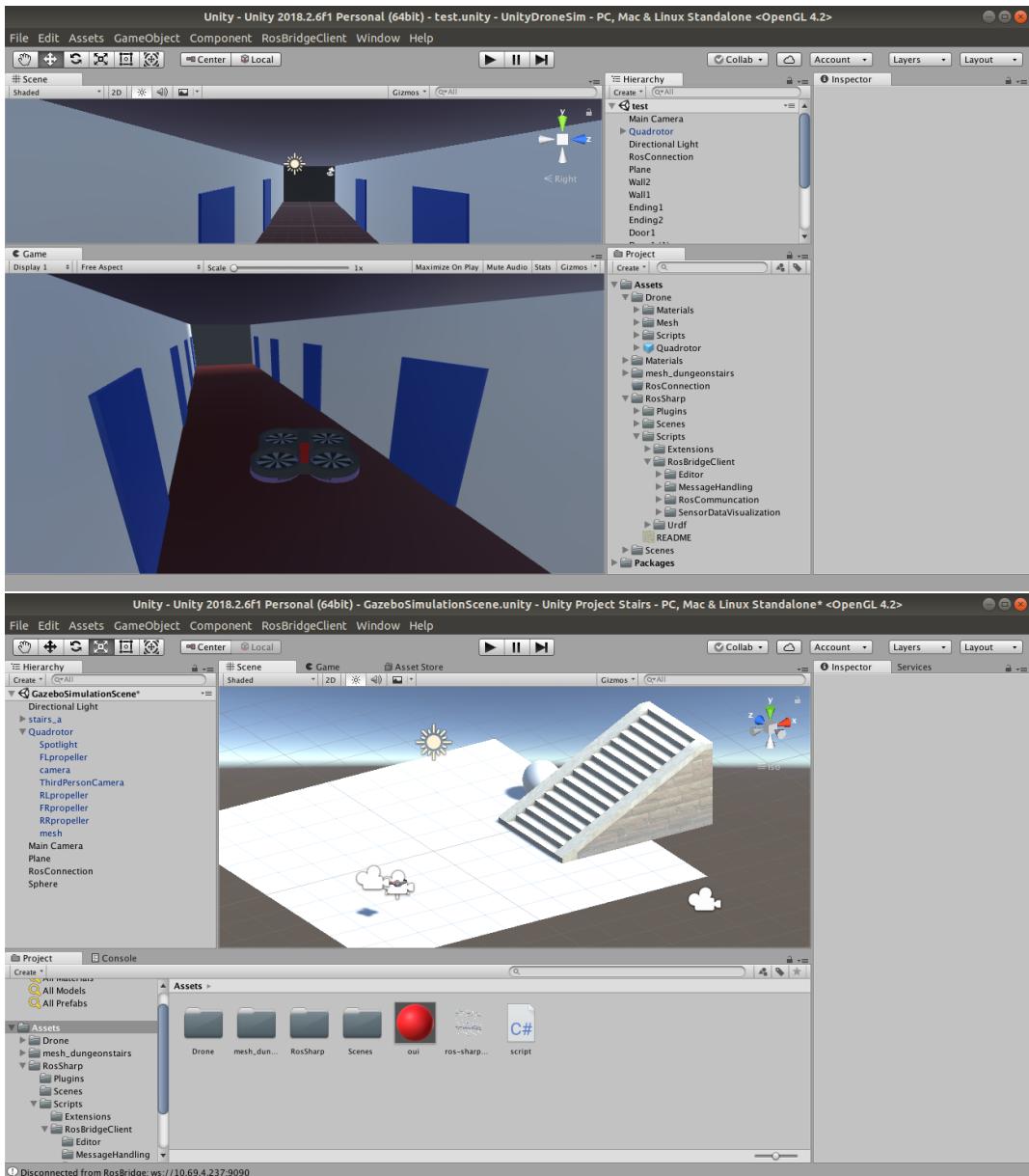


Figure 5 – Unity Simulation Scenes

2.2.3 Langage de développement et bibliothèque

ROS possède des API en Python et C++, nous avons ici choisi Python comme langage de développement. Nous avons utilisé en particulier les modules numpy et OpenCV afin de traiter les images reçues par les caméras du drone et scikit-learn pour identifier les éléments pertinents dans l'image.

3 Traitement d'image

Nous diviserons cette partie en trois différentes sections. En effet, nous avons utilisé des traitements différents en fonction de l'utilisation que nous en faisions après. La première section concerne le traitement d'image appliquée pour la navigation dans les couloirs avec notamment la détection de points de fuite. La seconde sera dédiée à celle appliquée aux escaliers avec la détection du centre de celui-ci et d'un trapèze modélisant sa forme. La troisième sera dédiée à la détection de porte grâce au flux optique que reçoit le drone.

3.1 Traitement d'image pour la partie Navigation dans un couloir

3.1.1 Introduction

Pour naviguer à l'intérieur d'un couloir, nous devons d'abord détecter le point de fuite. L'idée était de détecter toutes les lignes puis de calculer les points d'intersection de toutes les lignes 2 à 2. Ces points seront ensuite utilisés pour localiser le point de fuite dans le couloir.

3.1.1.1 L'algorithme en une seule image

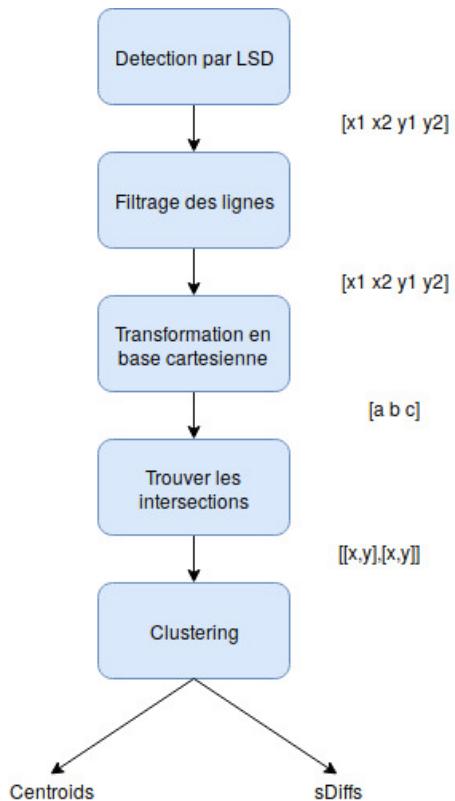


Figure 6 – L'algorithme

3.1.2 Détection des lignes

3.1.2.1 Utilisation de "Hough Line Transformation"

La première tentative a été d'utiliser la détection Canny Edge et la Transformation de Hough afin de détecter les lignes.

Les tests sur les images de couloir ont donné d'excellents résultats. Mais le problème était que lorsque nous appliquions cet algorithme sur un véritable flux vidéo, cela ne fonctionnait plus.

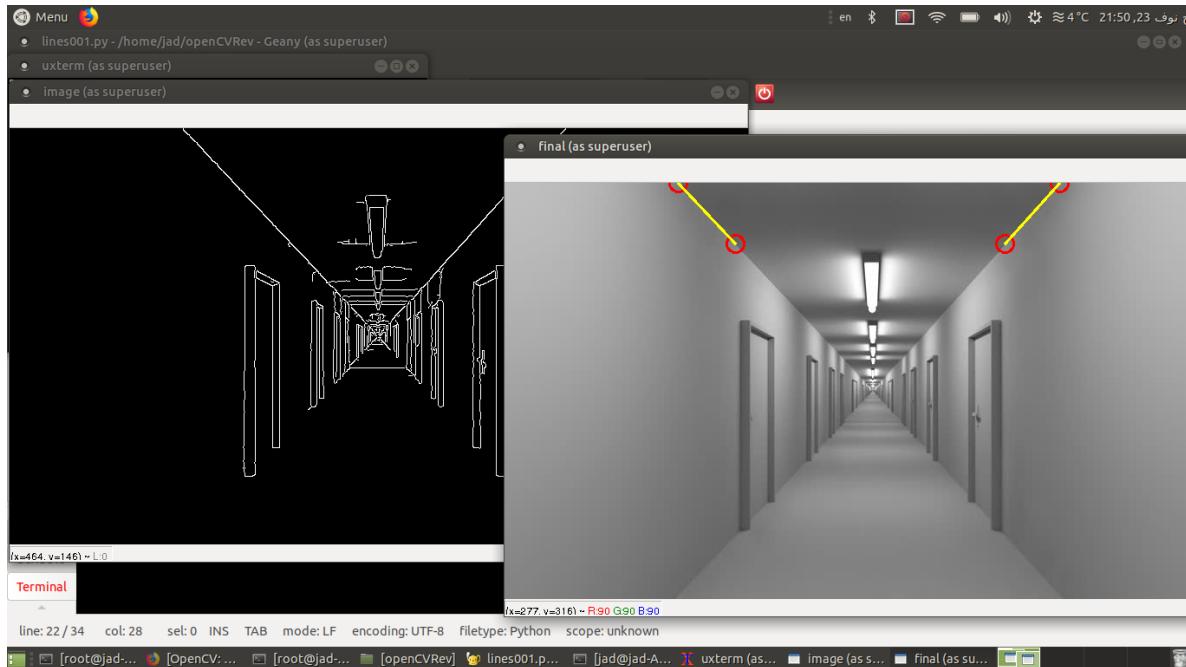


Figure 7 – Hough Line Transformation

3.1.2.2 Utilisation de Line Segment Detector(LSD)

La solution a consisté à remplacer la transformation de ligne de Hough par le détecteur de segment de ligne ou simplement "LSD". Les tests sur "LSD" ont donné de meilleurs résultats en streaming vidéo.

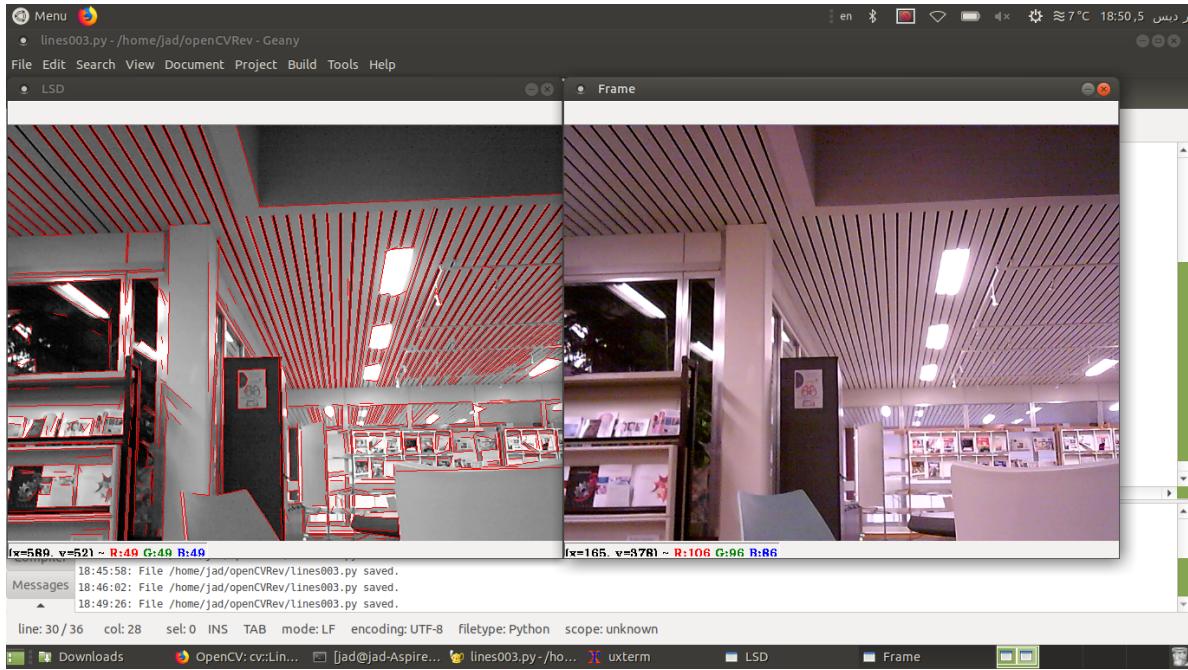


Figure 8 – LSD

Après avoir détecté les lignes, il faut appliquer un filtrage sur ces lignes pour ne garder que celles utiles aux étapes suivantes. On élimine les lignes courtes, verticales et horizontales.

3.1.3 Détection de point de fuite

3.1.3.1 Définition

Le point de fuite est le point où semblent converger les lignes parallèles décroissantes vues en perspective. Nous avons deux types de points de fuite. Dans cette partie, nous parlerons de point de fuite intérieur (Single Vanishing point).

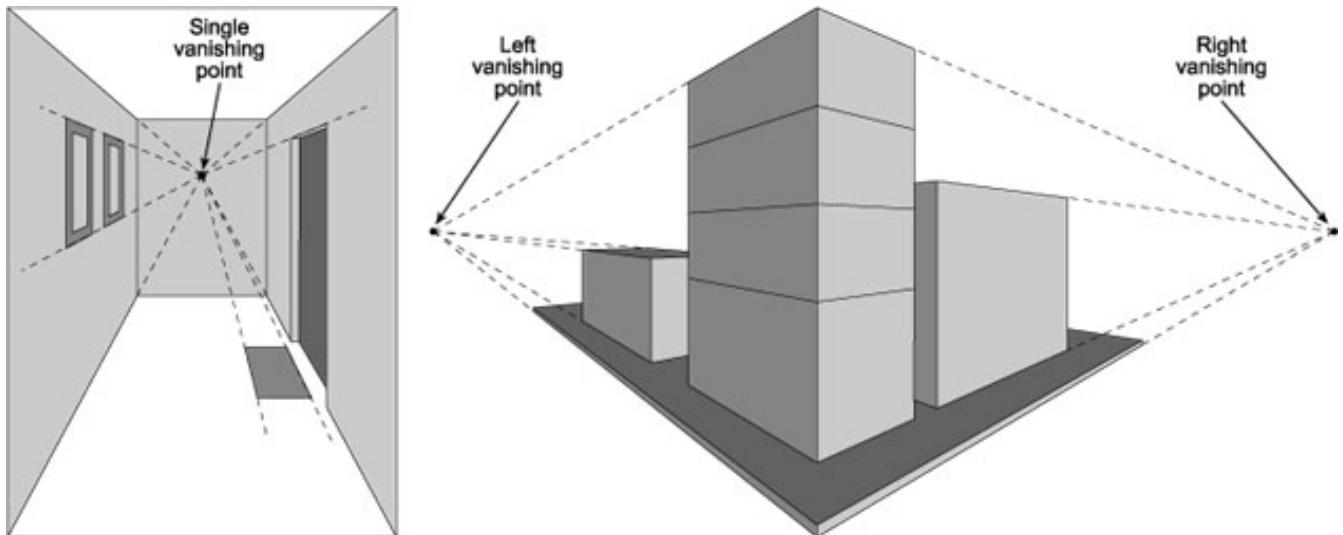


Figure 9 – Point de fuite

3.1.3.2 Représentation des lignes

Après avoir trouvé des lignes et éliminé les lignes verticales et horizontales, il est maintenant temps de trouver le point de fuite. Mais avant, nous devons choisir un formulaire pour représenter ces lignes. Avec Hough, les lignes de transformation sont présentées par des coordonnées cylindriques représentées par ρ et θ .

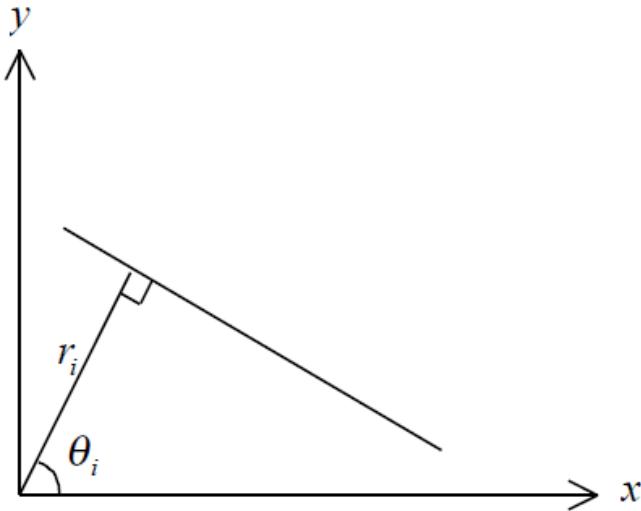


Figure 10 – Coordonnées Cylindriques

Mais comme nous n'avons pas utilisé cette méthode de détection de ligne, cette représentation n'a pas été adoptée.

L'autre représentation était $y = ax + b$ mais cette forme pose un problème évident pour les lignes verticales où a tend vers l'infini.

La représentation cartésienne ($ax + by + c = 0$) était la meilleure possible dans ce cas. La

représentation adoptée dans ce projet est la représentation cartésienne avec $vecv = (-b, a)$ le vecteur directeur de la ligne.

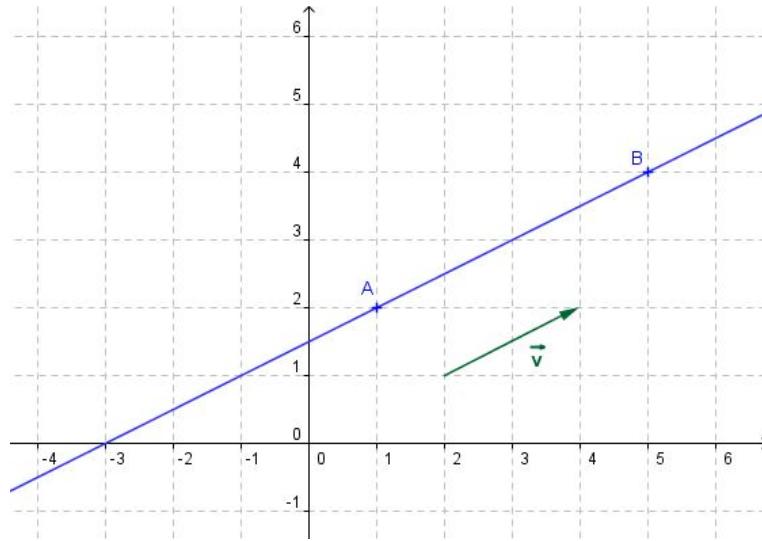


Figure 11 – Représentation cartésienne

3.1.3.3 Localisation du point de fuite

Dans ce projet, nous considérons que le drone se trouve dans un couloir et que le point de fuite se situe dans la zone où nous avons le plus grand nombre de lignes.

La fonction de recherche des intersections de lignes est la suivante :

```
def findGoodLines(self,allLines,length):
    allLines = np.reshape(allLines,[len(allLines),4])
    allLines[:,[1, 2]] = allLines[:,[2, 1]]
    diffLines = np.diff(allLines)
    diffLines = np.delete(diffLines,1,1)
    lengths = np.linalg.norm(diffLines, axis = 1)
    and_arr = np.logical_and(np.absolute(diffLines[:,0]) > 10,np.absolute(diffLines[:,1]) > 10)
    and_arr = np.logical_and(and_arr,lengths > length)
    #print final_and
    goodLines = allLines[and_arr]
    goodDiffs = diffLines[and_arr]
    #print goodlines[:,2]>180
    #on Y1 (60<Y1<180)
    and_arr2 = np.logical_and(goodLines[:,2]>60 , goodLines[:,2]<180)
    #on Y2 (60<Y2<180)
    and_arr3 = np.logical_and(goodLines[:,3]>60 , goodLines[:,3]<180)
    final_and = np.logical_not(np.logical_and(and_arr2, and_arr3))
    goodLines = goodLines[final_and]
    goodDiffs = goodDiffs[final_and]
    goodLines[:,[1, 2]] = goodLines[:,[2, 1]]
    return goodLines,goodDiffs
```

Figure 12 – Filtrage des lignes

Le LSD retournera un tableau numpy de forme $(X, 1, 4)$, X étant le nombre de lignes détectées. Cette fonction remodèle d'abord ce tableau numpy en $(X, 4)$, puis calcule le vecteur directeur, élimine les lignes verticales, horizontales, courtes et quelques lignes bruitées. Enfin, elle renvoie la

matrice des vecteurs directeurs et la matrice des lignes souhaitées. Et maintenant, il est temps de calculer les intersections de lignes, mais à ce stade, nous avons juste le a_s et b_s nous avons encore besoin du c_s . La première étape consiste à rechercher ces c_s , puis à rechercher les intersections de toutes les lignes, ce qui signifie résoudre le système linéaire de toutes les combinaisons de 2 lignes. Ce qui est fait par ces 2 fonctions :

```
def findCartMat(self,Points,Vects):
    #suppose points are given in [Xa,Ya] and the vectors are given in [Xu,Yu] we have that c = YaXu - XaYu = diff(XaYu,YaXu)
    #so we need to swap the Vects Matrix
    Vects[:,[0,1]] = Vects[:,[1,0]]
    cmat = np.multiply(Points,Vects)
    cmat = np.diff(cmat)
    return cmat

def findIntersec(self,abval,cval):
    abval[:,1] = np.multiply(-1,abval[:,1])
    cval = np.multiply(-1,cval)
    nbLines = cval.shape[0]
    index = np.fromiter(chain.from_iterable(combinations(range(nbLines), 2)), int)
    index = index.reshape(-1,2)
    A = abval[index,:]
    B = cval[index,:]
    interSec = np.linalg.solve(A, B).reshape(-1,2)
    and_arr1 = np.logical_and(interSec[:,0] > 0,interSec[:,1] > 0)
    and_arr2 = np.logical_and(interSec[:,1] < 320,interSec[:,0] < 240)
    return interSec[np.logical_and(and_arr1, and_arr2)]
```

Figure 13 – Trouver les intersections

3.1.3.4 Clustering

Après avoir trouvé des intersections de lignes, nous devons trouver la zone de l'image où nous avons la plus grande densité de points d'intersection et le centroïde de cette zone sera le point de fuite. Pour trouver la zone avec la plus grande densité de points, nous allons utiliser dans ce projet l'algorithme DBSCAN Clustering.

" Density-based spatial clustering of applications with noise or DBSCAN is an algorithm proposed by Martin Ester, Hans-Peter Kriegel, Jörg Sander and Xiaowei Xu in 1996. It is a density-based clustering algorithm : given a set of points in some space, it groups together points that are closely packed together (points with many nearby neighbors), marking as outliers points that lie alone in low-density regions (whose nearest neighbors are too far away). DBSCAN is one of the most common clustering algorithms and also most cited in scientific literature."

-Wikipedia-

Dans ce projet, nous avons utilisé la fonction DBSCAN de sklearn.

```

def findCentroid(self, intersec):
    if intersec.shape[0] == 0:
        return 0,0
    if intersec.shape[0] == 1:
        return 1,intersec[0]

    db = DBSCAN(eps=20, min_samples=2).fit(intersec)
    db_labels = db.labels_
    labels, counts = np.unique(db_labels[db_labels>=0], return_counts=True)
    maxlabel = labels[np.argsort(-counts)[:1]]
    goodPoints = intersec[db_labels == maxlabel]
    centroid = np.mean(goodPoints, axis=0)
    return 1,centroid

```

Figure 14 – Clustering

Après détection du point de fuite, l'abscisse de ce dernier est publiée sur le topic */centroids* et utilisé pour calculer une autre information qui sera utile pour la commande.

3.1.3.5 Filtrage du flux des valeurs

Lorsque nous traitons des images sur un flux d'images compressées, nous pouvons constater que les lignes et les intersections, puis le point de fuite se déplacent beaucoup. Le signal est bruité. Et comme on utilise la position du centroïde comme entrée pour la commande drone, obtenir une entrée bruitée rendra le drone instable.

La solution consiste à appliquer un filtre de moyennage du premier ordre sur les données de centroïde à l'aide de la formule suivante : $\hat{y}[n] = \hat{y}[n - 1] + \alpha(y[n] - \hat{y}[n - 1])$ Après avoir réglé α , nous pouvons obtenir un bon résultat et les valeurs du centroïde seront lisses.

Vous pouvez voir les résultats dans la figure suivante

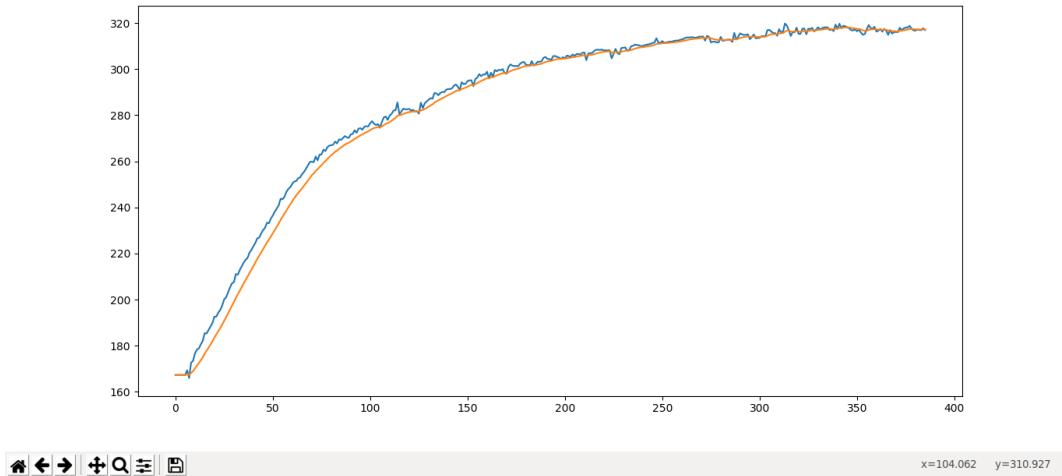


Figure 15 – Filtrage

3.1.3.6 La différence entre les pentes

Après avoir trouvé le point de fuite, l'image sera divisée de la manière expliquée dans cette figure.

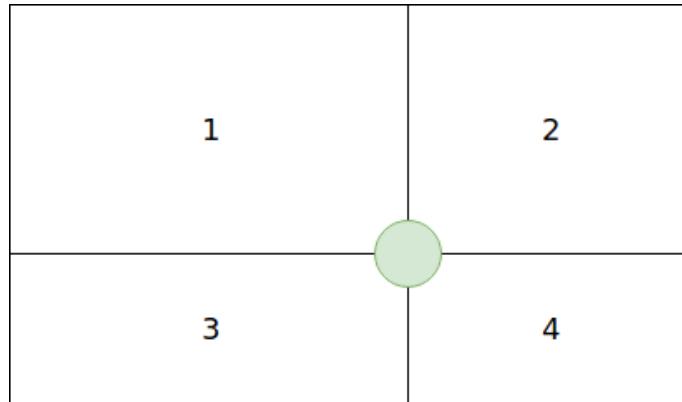


Figure 16 – Division de l'écran

Le rectangle extérieur représente l'image OpenCV affichée par le noeud et le cercle vert représente le point de fuite détecté. Après avoir détecté le point de fuite, nous divisons l'image. Nous sélectionnons les lignes qui sont entièrement ou partiellement dans les zones 3 et 4 qui traversent presque le point de fuite et nous calculons la moyenne des pentes. Ces données sont utilisées pour envoyer une commande au drone.

3.1.4 Résultats

3.1.4.1 Simulation

Ici, nous allons vous montrer quelques résultats des simulations dans un "asset" Unity3D de sous-sol (Basement And Sewerage Modular Location by Sébastien Marevoy)

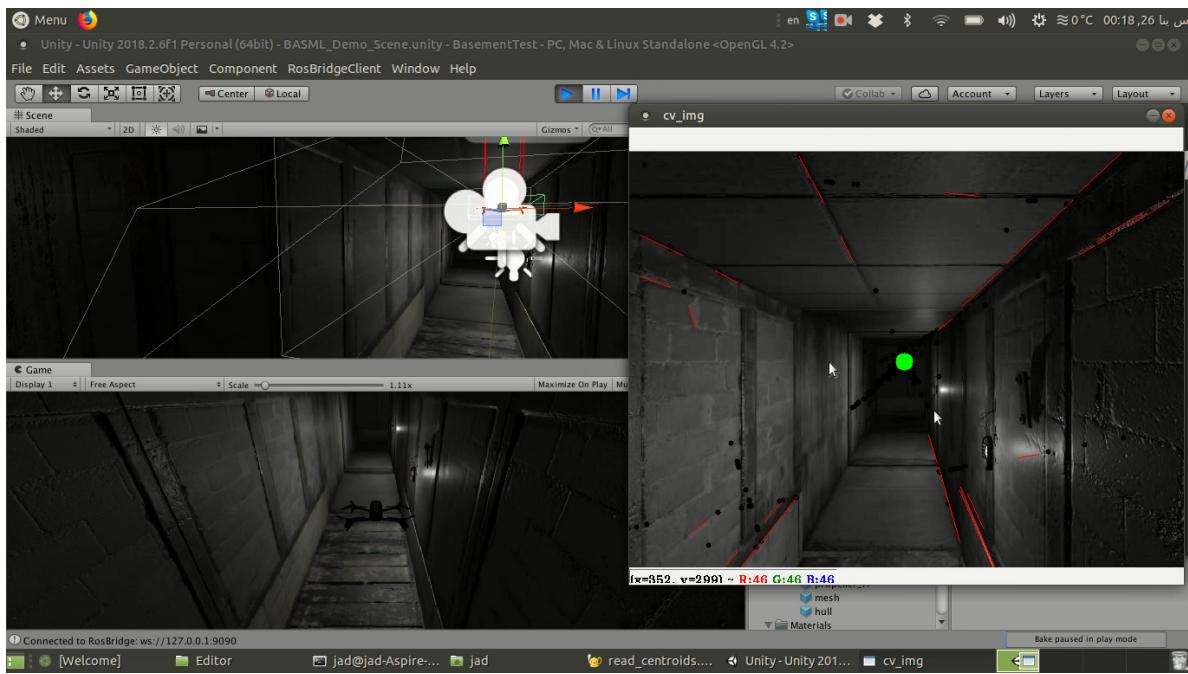


Figure 17 – Test dans un sous-sol

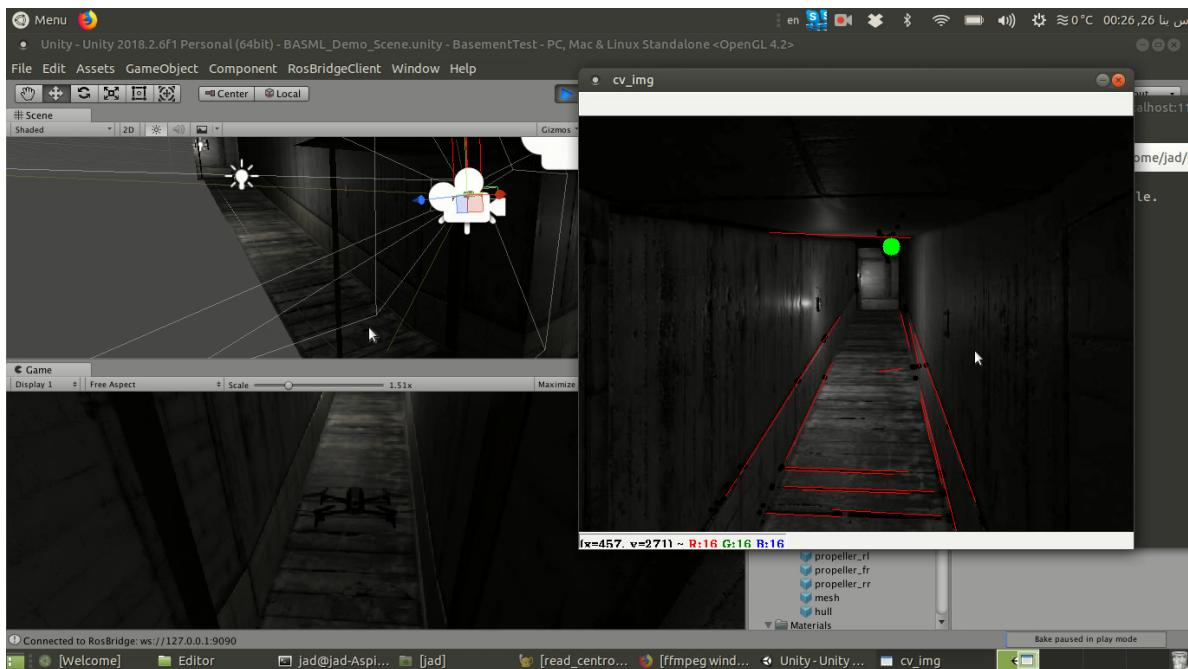


Figure 18 – Test dans un sous-sol

3.1.4.2 Vrai Drone

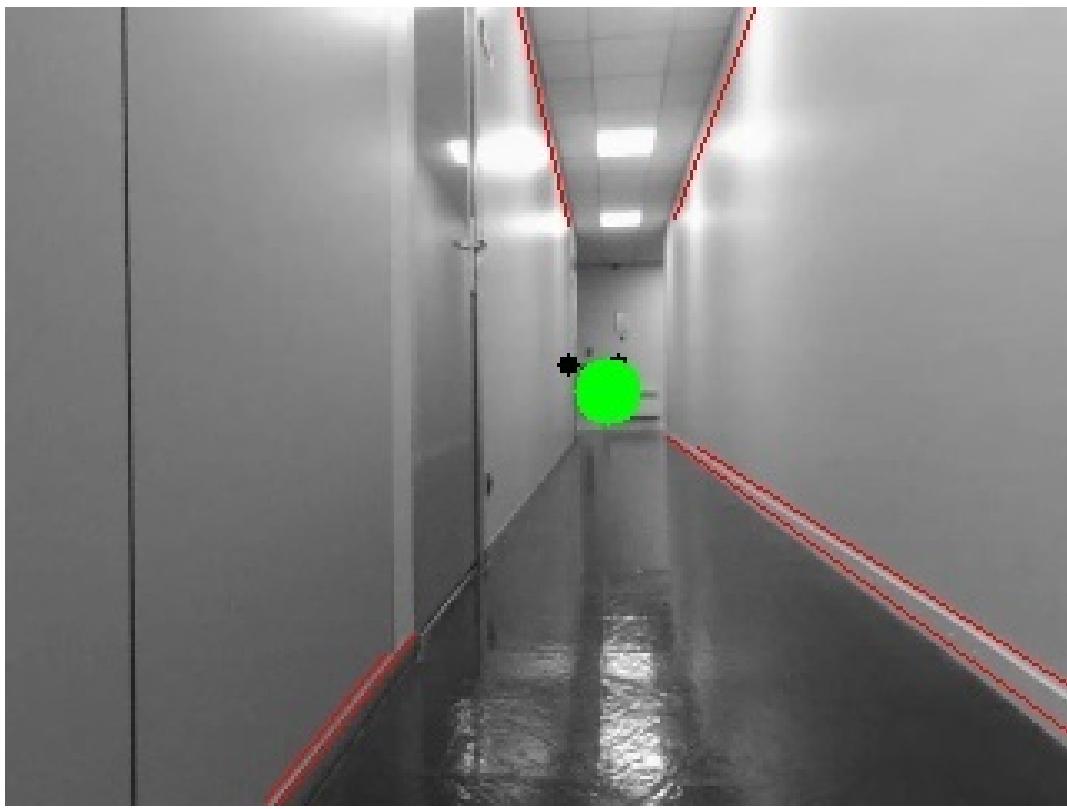


Figure 19 – Test dans l’École

Cette image montre les résultats d'un test réel de l'algorithme de détection du point de fuite dans les couloirs de l'école.

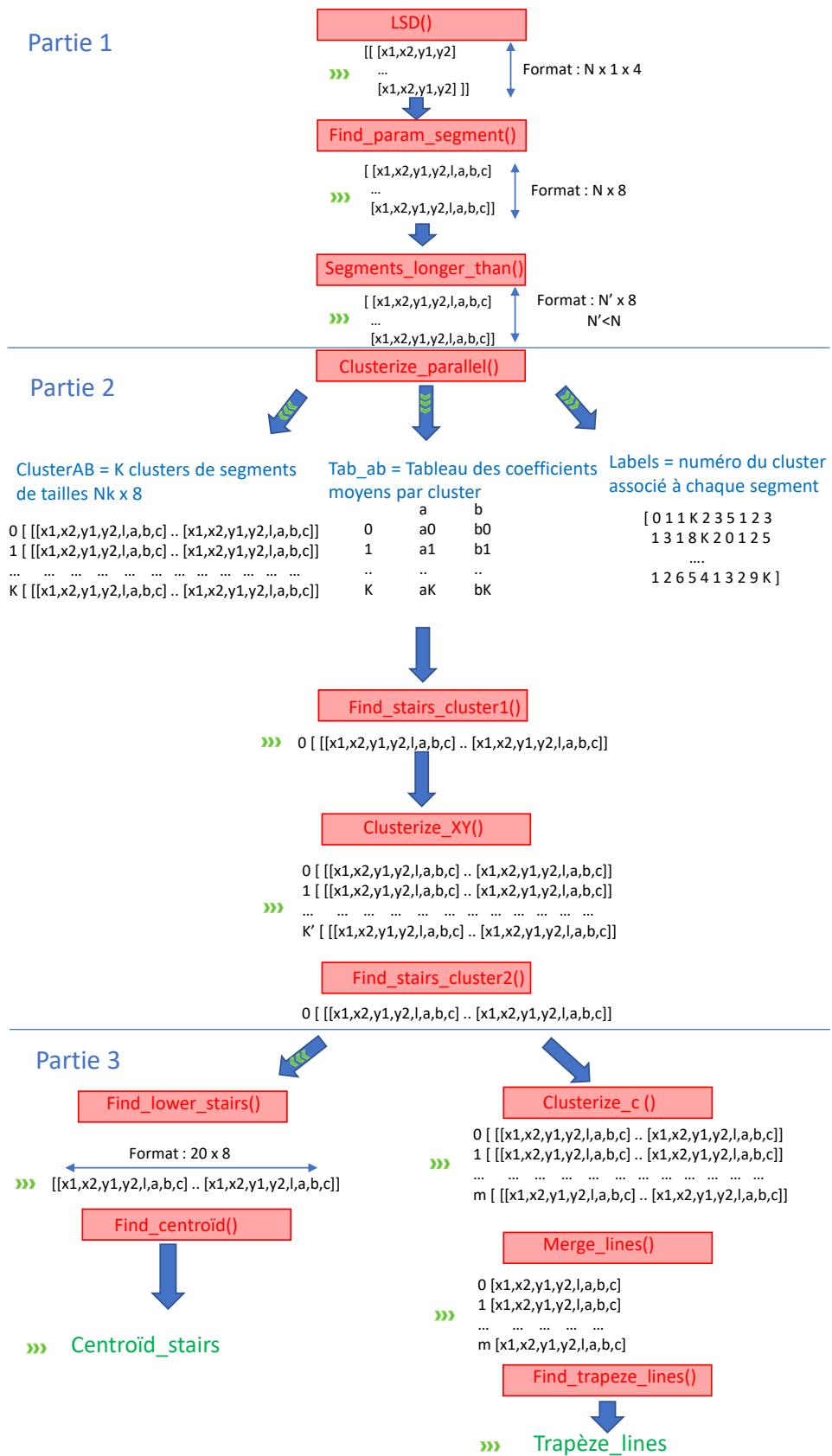
3.2 Traitement d'image de la partie Escalier du projet

3.2.1 Introduction

Nous nous intéressons dans ce projet à la détection d'un escalier. Quelles sont ses caractéristiques ? Qu'est ce qui le différencie d'un autre objet ?

Grâce au travail des mathématiciens italiens du *XVII^{ème}* siècle sur la projectivisation de \mathbb{R}^3 , nous savons extraire de l'image les informations sur la structure tridimensionnelle de ces escaliers. Tout d'abord un escalier possède une structure particulière, en forme de trapèze, lorsqu'il est de face (ce qui est dû à la profondeur). Il contient de nombreux segments, dus aux marches. Ces segments sont quasiment parallèles, ou tout du moins possèdent un angle avec l'horizontale quasiment identique (voir figure 20).

Lorsque l'on voit un escalier de biais, les marches possèdent un angle avec l'horizontale, mais les propriétés énoncées ci-dessus perdurent. En effet, l'escalier peut toujours être modélisé par un ensemble de segments, possédant un angle avec l'horizontale quasiment identique (voir figure 20). Nous utiliserons par la suite ces caractéristiques afin de localiser un escalier sur une image. Nous allons dans un premier temps détecter les segments d'une image (partie 1). Nous allons ensuite traiter les segments et les regrouper en fonction de leurs angles avec l'horizontal, grâce à leurs vecteurs directeurs puis extraire de cet ensemble celui correspondant à l'escalier (partie 2). Cela grâce à un outil, appelé clustering. Nous allons enfin extraire les données utilisables afin de pouvoir diriger le drone (partie 3). Le traitement d'image est résumé sur la figure page suivante.



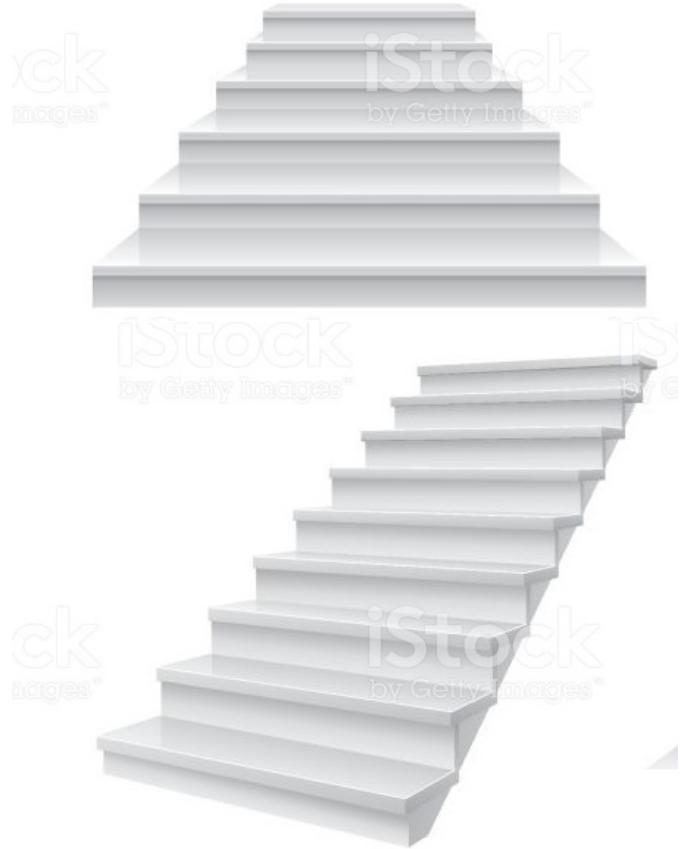


Figure 20 – Escalier de face et de coté

3.2.2 Preprocessing

3.2.2.1 Détection des lignes avec LSD

La première étape a été de détecter les segments d'une image. Pour cela nous avons utilisé la bibliothèque LSD (Lines Segment Detector) fournie par python.

Nous avons tout d'abord transformé l'image en nuances de gris afin de pouvoir la traiter plus facilement, puis nous avons initialisé les paramètres par défaut et enfin nous avons récupéré les segments.

```
1 img = cv2.imread("escaliers2.jpg",0)
2 gray = img
3 lsd = cv2.createLineSegmentDetector(0)
4 segmentsArray = lsd.detect(gray)[0]
```

Listing 1 – Python LSD



Figure 21 – LSD

3.2.2.2 Extraction des segments intéressants

Le premier problème que nous avons rencontré était que l'image reçue n'était pas parfaite. En effet, on peut voir apparaître de nombreux petits segments semblables à des points qu'il a fallu supprimer pour pouvoir traiter l'image de façon correcte. Pour faire cela nous avons dû extraire les données fournies par LSD et les traiter afin de ne garder que les plus grandes lignes.



Figure 22 – Détermination des paramètres d'un segment

Détermination des paramètres d'un segment Nous avons tout d'abord supprimé la dimension supplémentaire apportée par LSD qui n'était d'aucune utilité et qui nous a causé de nombreux problèmes de format.

Nous avons ensuite cherché à déterminer les paramètres des lignes que sont les coefficients directeurs et leur longueur pour pouvoir les traiter plus facilement. En effet, dans un premier temps nous avons cherché à supprimer les lignes trop courtes (utilisation de la longueur des segments). Puis nous nous sommes rendus compte par la suite de l'utilité de toutes les caractéristiques d'un segment, c'est à dire, les paramètres a, b, c de l'équation cartésienne d'une droite $ax+by+c=0$. Car un segment n'est qu'un tronçon de droite. Nous rappelons que le vecteur directeur d'une droite est :

$$\overrightarrow{V_{dir}} \begin{vmatrix} -b \\ a \end{vmatrix} \text{ et que le vecteur normal est : } \overrightarrow{V_{norm}} \begin{vmatrix} a \\ b \end{vmatrix}$$

Nous avons donc transformé le tableau fourni par LSD en un autre tableau de dimension $N \times 8$ qui regroupe les coordonnées des extrémités du segment (x_1, y_1, x_2, y_2), la longueur l et les coefficients directeurs (a, b, c) de la droite correspondante. Nous avons par ailleurs normalisé les paramètres a et b afin que le vecteur V_{norm} soit unitaire, ceci afin de faciliter son utilisation par la suite et de pouvoir comparer les segments entre eux.

Suppression des lignes inutiles Nous avons ensuite décidé de supprimer les lignes très petites (inférieures à l_{min}) afin que ce bruit ne perturbe pas le traitement que nous allons faire par la suite.

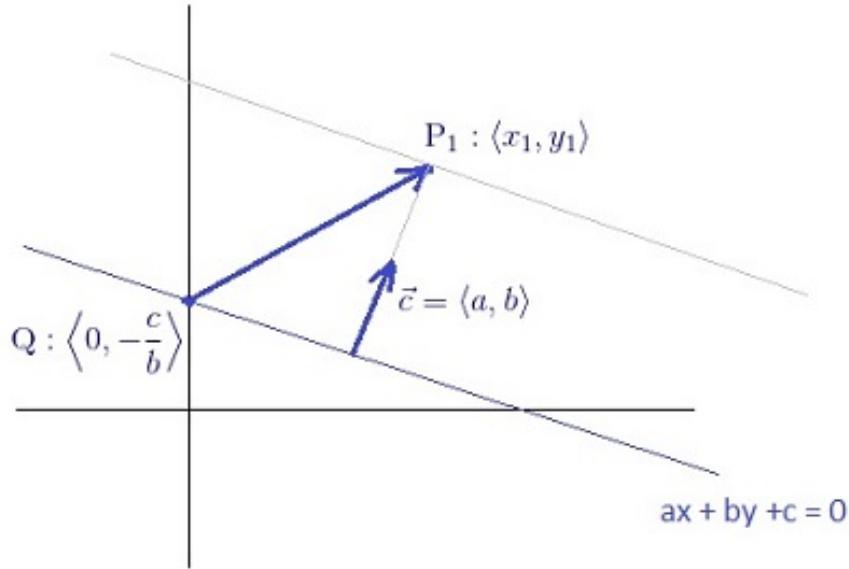


Figure 23 – Droite et ses coefficients

Nous avons pris lmin=30 pixels. Voici la fonction permettant de faire ce traitement.



Figure 24 – Suppression des lignes trop courtes

Nous avons utilisé les outils mis à disposition par numpy afin d'effectuer des comparaisons directement sur des tableaux. Cela évite de faire des boucles et ainsi de réduire la complexité en temps de calcul et en mémoire. Nous avons essayé de respecter cette logique tout au long de notre projet. En effet la rapidité de traitement est essentielle pour le pilotage du drone, si le traitement est trop long, les commandes ne seront pas données en temps réel ce qui ne pourra que conduire à la catastrophe.

3.2.3 Extraction des segments de l'escalier

Nous allons dans cette partie, expliquer comment nous avons réussi à détecter un escalier au sein d'une image. Comme nous l'avons énoncé dans l'introduction, un escalier est caractérisé par 3 choses :

1. un nombre important de segments
2. des segments proches géographiquement
3. des segments possédant des angles avec l'horizontale "proche".

Pour les rassembler nous avons utilisé un outil appelé *cluster* qui permet de regrouper des objets (vecteurs dans notre cas) possédant des caractéristiques proches (vecteurs directeurs).

3.2.3.1 Explication du clustering

Nous allons utiliser le module DBscan de la bibliothèque `sklearn.cluster` afin de clusterizer un ensemble de vecteurs à partir de certaines caractéristiques que sont les coordonnées du vecteur normal. Ainsi nous arrivons à regrouper les lignes possédant des vecteurs normaux proches, c'est à dire les segments parallèles. Afin d'effectuer cela, nous avons normalisé tous les vecteurs, pour qu'ils aient une norme unitaire. Ainsi tous les paramètres a et b du vecteur (a,b) seront compris entre 0 et 1. En les affichant, avec comme abscisses b et en ordonnées a , tous les vecteurs seront regroupés sur un cercle, de centre $(a=0, b=0)$ et de rayon 1.

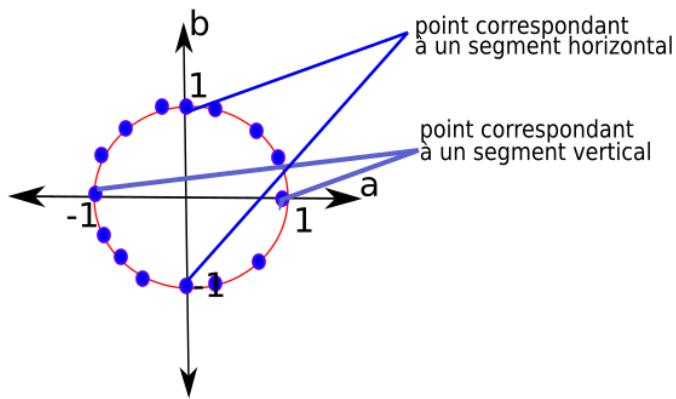


Figure 25 – Normalisation des vecteurs directeurs

Enfin, nous pouvons regrouper ces vecteurs (correspondant aux segments de l'image) grâce au clustering, avec comme paramètre ϵ_{AB} qui correspond à la distance maximale entre les coefficients de deux vecteurs. Comme deux points diamétriquement opposés sur ce cercle correspondent à une même droite (en effet, une droite ne possède pas de sens mais seulement une orientation), on multiplie le tout par le signe de b afin d'avoir un b à norme toujours positive. En effet, un escalier est globalement horizontal, et correspondra idéalement à un couple $(a, b) = (0, 1)$, quelles que soient l'écart de l'angle de l'escalier par rapport à l'horizontale, il ne sera jamais vertical. On ne se retrouvera donc jamais avec deux points correspondant à un même cluster théorique qui se retrouveraient à deux extrémités différentes du demi-cercle (ce qui aurait pour conséquence que le clustering ne les regroupe pas puisque ces points seraient considérés comme génétiquement éloignés). Ce choix est donc parfaitement adapté à la détection d'escaliers.

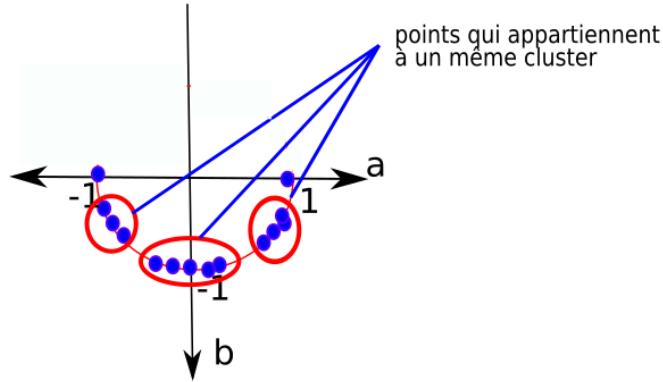


Figure 26 – Fonctionnement du clustering

Cela nous permet de regrouper tous les segments reçus de lsd en clusters de segments regroupant tous les segments possédant des vecteurs normaux proches.

3.2.3.2 Formation des clusters

Une fois les clusters formés, nous avons modifié notre fonction afin qu'elle puisse renvoyer :

1. un tableau contenant les a et b moyens de chacun des clusters.
2. les clusters sous forme de série (cela permet de regrouper les segments correspondant à un même cluster dans une même ligne, tout en ayant une taille de lignes variable)
3. les labels (qui est un tableau de la même taille que les segments et qui associe à chacun des segments son numéro de cluster)

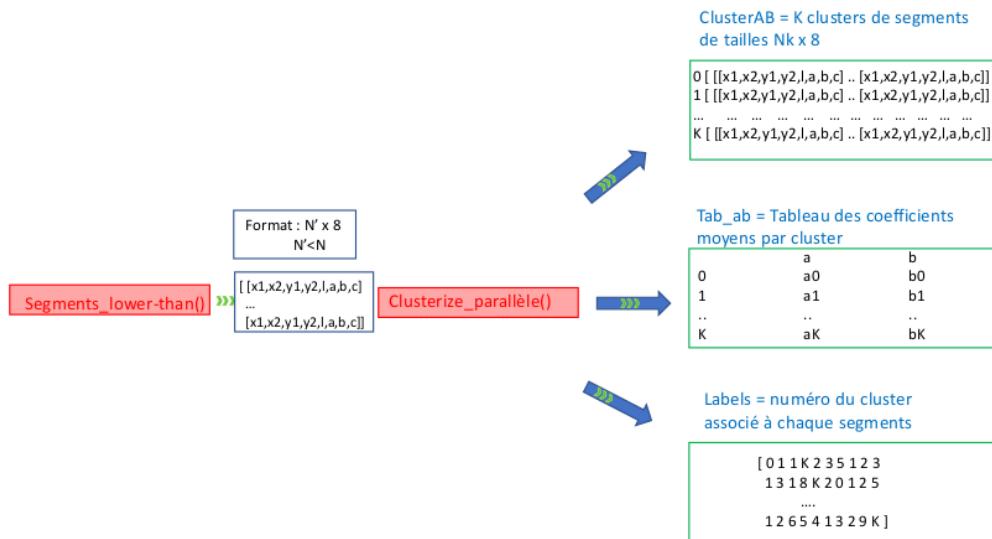


Figure 27 – Regroupement des segments par (a,b) semblables

```

1 0 [[1177.186, 403.24377, 1207.8324, 0.5076154, 4...
2 1 [[1290.8519, 305.3794, 1540.0621, 134.89774, 3...
3 2 [[1200.4152, 736.87756, 1199.7874, 685.63525, ...
4 3 [[1410.3212, 913.42053, 1442.9789, 916.9695, 3...
5 4 [[1372.3132, 994.0648, 1402.6493, 914.67084, 8...
6 5 [[1225.7207, 1101.9193, 1256.5162, 1035.3658, ...
7 6 [[969.469, 1297.9922, 1031.8431, 1292.7458, 62...
8 7 [[1211.7926, 2529.1843, 1273.4171, 2502.551, 6...
9 8 [[1892.135, 2591.0732, 1958.8455, 2508.7085, 1...

```

Listing 2 – Exemple réel de Série de cluster

	a	b
0	0.995710	0.087217
1	0.674166	0.736502
2	-0.999638	0.022488
3	-0.393444	0.905955
4	0.937522	0.347788
5	0.894359	0.446694
6	0.150150	0.987573
7	0.391043	0.920174
8	0.787893	0.615707

Listing 3 – Exemple réel du tableau des coefficients (a.b) par cluster

```

1 [ 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 0 1 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 1 1 0
  0 0 1 1 0 0 1 0 1 0 0 1 2 0 0 1 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 1 1 1 1
  0 3 4 3 3 0 0 1 3 3 3 3 3 3 1 3 1 1 4 0 0 3 5 3 1 3 1 2 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  3 3 1 1 0 1 3 0 1 3 3 3 3 2 1 1 0 1 3 1 1 1 3 1 1 1 2 5 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  0 1 1 1 3 1 0 3 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1 1 3 1 1
]
```

Listing 4 – Exemple de labels

Chaque segment possède un label qui correspond au cluster qui lui est associé. Ainsi, on peut aisément classer les segments par cluster. De plus, le fait de renvoyer les a et b moyens nous permet de déterminer l'angle moyen des segments du cluster avec l'horizontal. En effet, $\theta = \arctan \frac{b}{a}$ est l'angle que fait le segment possédant les coefficients a et b avec l'horizontale.

3.2.3.3 Affichage des clusters

Nous avons ensuite écrit une fonction qui permet d'afficher ces clusters, et qui associe à chaque cluster une couleur en fonction de son angle moyen. En effet, à un cluster est associé un angle (par rapport à l'horizontale) et cet angle correspond à l'angle moyen de tous les segments appartenant à ce cluster.

Afin de faciliter l'affichage lorsque l'on utilisera cette fonction avec une vidéo, nous avons créé une image openCV. Sur cette image nous avons dessiné des axes (statiques), et nous avons affiché les points correspondant aux segments (évoluant en temps réel). Ainsi nous pouvons voir comment les directions des segments détectés évoluent lorsque l'on bouge notre caméra. Par ailleurs, nous avons établi une correspondance entre la couleur des points dessinés sur cette image et la couleur des segments dessinés sur l'escalier afin de faciliter la visualisation.

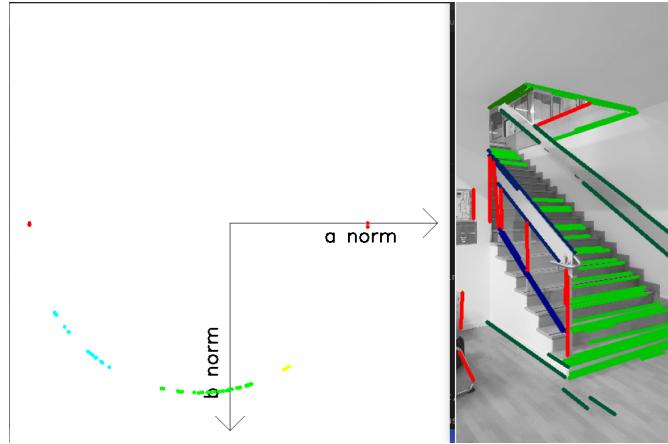


Figure 28 – Affichage des clusters montrant la couleur des segments en fonction de l’angle moyen du cluster

Nous avons par ailleurs, regroupé les clusters possédant des a identiques. En effet un segment possédant comme vecteur normal $(1, 0)$ est identique à un segment possédant le vecteur normal $(-1, 0)$.

3.2.3.4 Détection des segments de l’escalier

Il nous reste maintenant à isoler les segments correspondant à l’escalier. Pour cela, nous avons gardé le plus grand cluster, possédant un angle compris entre $\frac{\pi}{3}$ et $-\frac{\pi}{3}$. En effet, sélectionner le cluster le plus grand ne suffisait pas, car nous nous retrouvions souvent avec des clusters correspondant à des droites verticales. Certes, l’escalier est caractérisé par son grand nombre de segments parallèles, mais non avons imposé que la direction de ces droites ne soit pas trop proche de la verticale. La série des clusters ordonne les clusters par nombre décroissant d’éléments. On parcourt donc la série de cluster du plus fourni au moins fourni en nous arrêtant au premier plus gros cluster possédant un angle satisfaisant par rapport à l’horizontale.

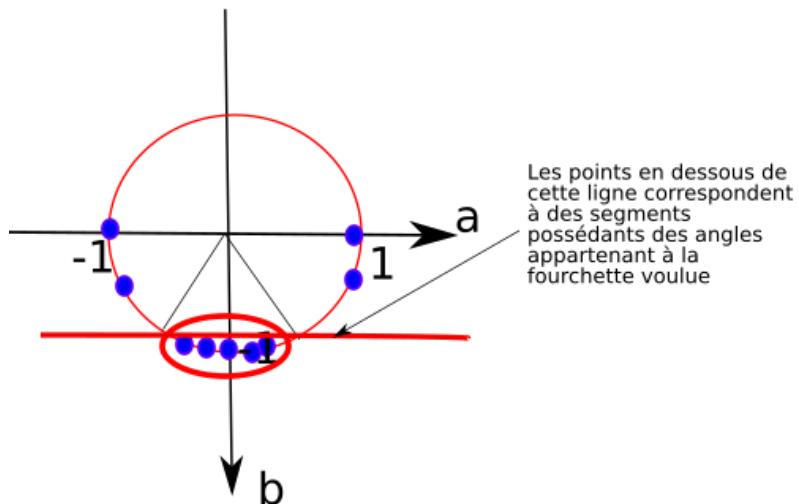


Figure 29 – Schéma montrant la sélection du cluster correspondant aux segments de l’escalier

Nous obtenons alors tous les segments possédant le bon angle.

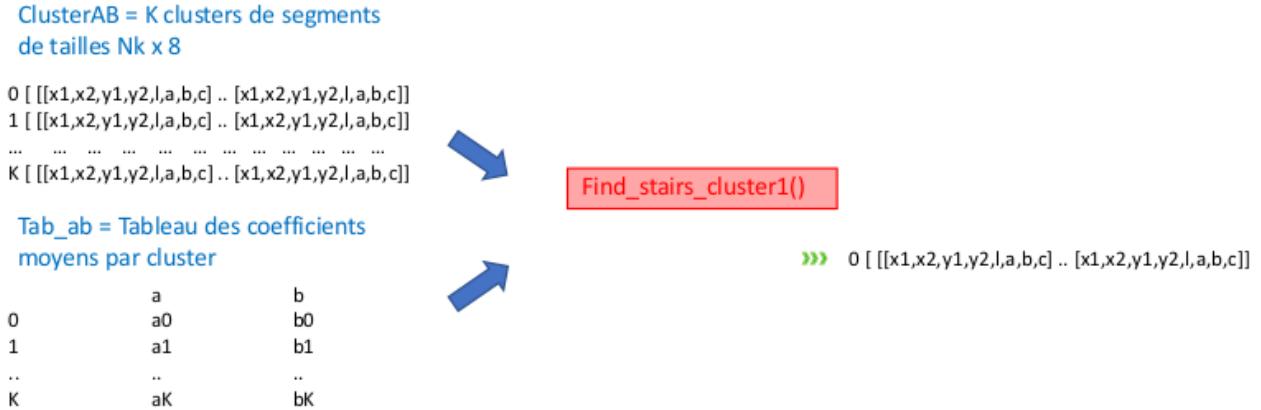


Figure 30 – Schéma de la sélection du cluster de l'escalier

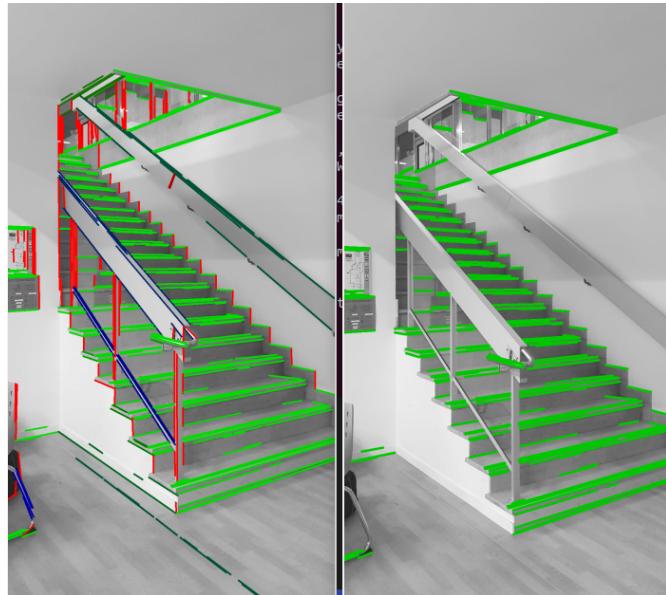


Figure 31 – Figure montrant la sélection du cluster contenant les segments de l'escalier

3.2.3.5 Isolement de l'escalier

Enfin nous avons utilisé notre 3^{ème} caractéristique de l'escalier. En effet un escalier est un regroupement de lignes possédant le même angle mais qui sont proches sur l'image. Nous avons donc utilisé une fonction permettant d'effectuer un regroupement sur les coordonnées des segments dans l'image. Pour cela nous avons extrait le milieu de chacun des segments et nous les avons clusterisé sur ce critère. Cela fonctionne plutôt bien.



Figure 32 – Isolement de l’escalier



Figure 33 – Affichage des segments avant la sélection

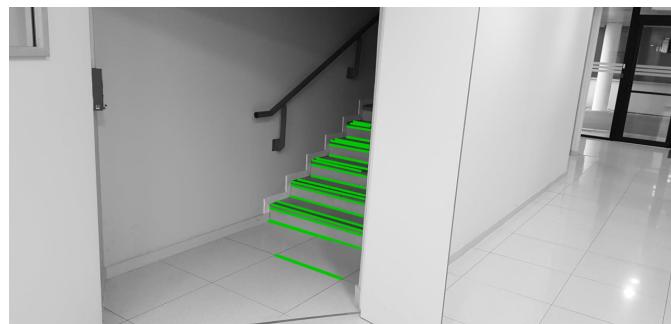


Figure 34 – Affichage des segments de l’escalier

3.2.4 Extraction de l’information de l’escalier

Nous allons maintenant nous intéresser à l’extraction de l’information contenue dans l’escalier afin de pouvoir piloter le drone. Nous allons diviser cette section en deux parties. La première partie consistera à détecter l’escalier puis à diriger le drone vers celui-ci. La seconde consistera à monter l’escalier.

3.2.4.1 Comment aller vers l’escalier ?

Notre première approche a été de détecter le milieu de l’escalier afin de le donner comme cible au drone. Nous avons remarqué que cela fonctionnait bien dans le cas de certains escaliers, type smartroom (voir figure-ci-après).

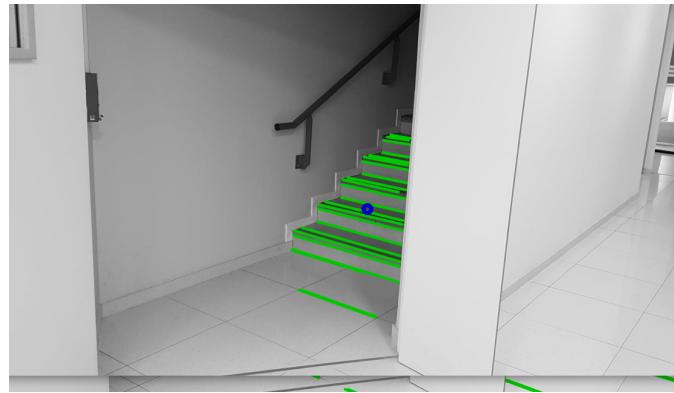


Figure 35 – Affichage du centre de l'escalier : Escalier type 2

Cependant, lorsque tout l'escalier est visible, même lorsque l'on est sur le côté, cette solution pose problème.



Figure 36 – Affichage du centre de l'escalier : Escalier type 1

Nous avons donc adopté comme solution, de viser le milieu des segments du bas de l'escalier. Pour cela nous avons extrait les segments possédant les plus grand y. En effet, l'origine des axes sur une image est en haut en gauche. Ainsi les segments de plus grands y seront les lignes du bas et celle de plus grand x celles les plus à droites de l'image. Cela fonctionne pour les 2 types d'escalier.



Figure 37 – Sélection des 20 marches les plus basses de l'escalier (4 dans l'algorithme final)

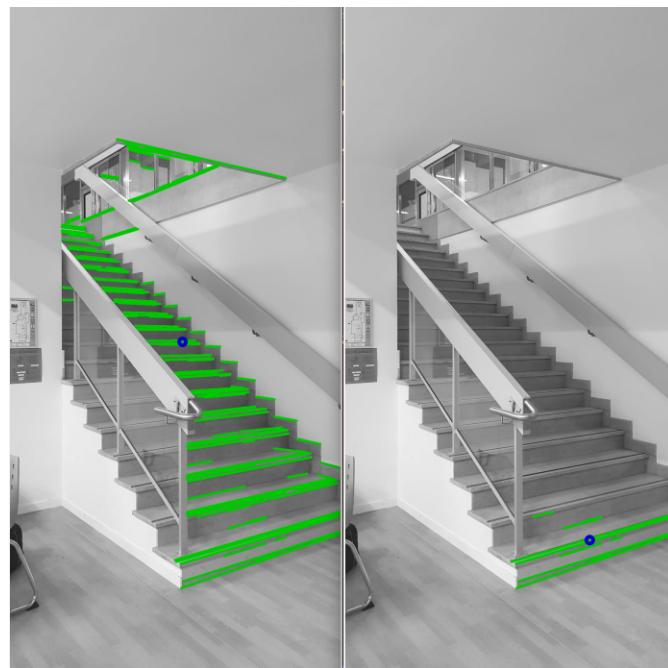


Figure 38 – Affichage du centre de l'escalier avant et après la sélection des bons segments : Escalier type 1

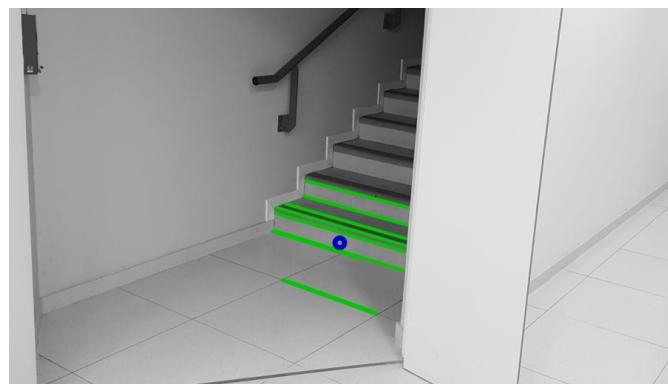


Figure 39 – Affichage du centre de l'escalier après la sélection des bons segments : Escalier type 2

Voyant que cela fonctionnait de façon satisfaisante, nous avons appliqué cette méthode afin de pouvoir monter l'escalier. C'est à dire, une fois bien en face de l'escalier, utiliser ce point de référence afin de rester au milieu de l'escalier. Cependant, les lignes détectées variant sans cesse, le point changeait tout le temps de position. Utiliser cette information bruitée ne pouvait pas fonctionner. Nous avons donc décidé d'utiliser une autre technique afin de lui faire monter l'escalier. Il fallait en

effet utiliser le plus de lignes possible afin de réduire la sensibilité de la détection et améliorer sa stabilité. La méthode utilisant un trapèze.

3.2.4.2 Méthode du trapèze

Afin de piloter le drone en utilisant le trapèze modélisant l'escalier, nous allons tout d'abord effectuer quelques traitements. En effet, la détection des lignes n'étant pas parfaite, de nombreuses lignes sont représentées par plusieurs segments. Il est alors impossible de dessiner le trapèze à partir de cela. Afin de fusionner ces segments, il a tout d'abord fallu les rassembler. Pour faire cela nous avons utilisé le 3^{ème} paramètre des lignes qui est le paramètre c de l'équation $ax+by+c=0$. Ensuite, nous avons fusionné ces segments avec une fonction merge() qui rassemblait celles ayant les bonnes caractéristiques. Enfin, nous en avons extrait le trapèze de deux façons. En effectuant des régressions linéaires classiques et en utilisant une SVM.

Extraction des bonnes lignes avec le paramètre c

Afin de reconstituer les marches complètes à l'aide des bons segments, nous avons utilisé le paramètre c. Ce paramètre est en quelques sortes, l'offset de la ligne. Deux lignes avec des mêmes paramètres (a,b) seront parallèles. Mais si elles possèdent un c différent, par exemple $c1 > c2$. Le segment 1 sera parallèle au premier, mais au-dessus de celui-ci. (voir figure 23) Ainsi, les segments correspondant à une même ligne de l'image, seront les segments possédant un vecteur (a,b) très proche, mais aussi un offset proche.

Nous avons donc clusterisé avec le paramètre c tous les segments de l'escalier. Chaque cluster, rassemblera alors plusieurs segments correspondant à une même ligne réelle.

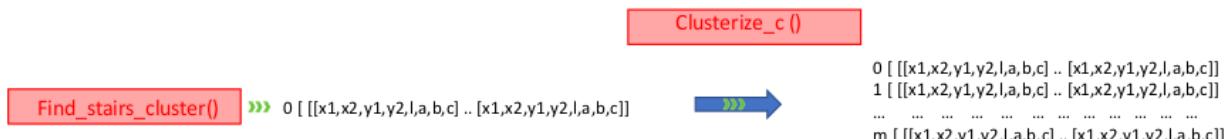


Figure 40 – Regroupement des segments par coefficients c

Fusion des segments correspondant à une même ligne

Une fois cela effectué, il reste à fusionner toutes les lignes du même cluster. Une petite subtilité était que quand une ligne était détectée "parfaitement", c'est à dire que cette ligne n'était représentée que par un seul segment, son label était de -1. Autrement dit, toutes les lignes bien détectées, sont classées comme "mauvaises lignes" (i.e. hors cluster). Il a donc fallu les extraire puis fusionner le tableau des lignes fusionnées, et de celle "parfaitement détectées".

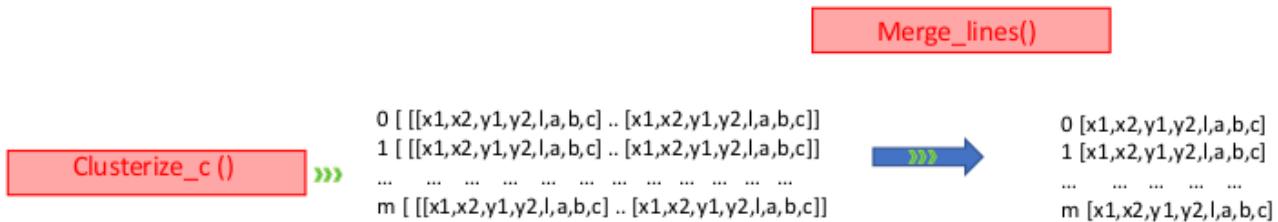


Figure 41 – Fusion des lignes

Voici un exemple des segments, avant et après la fusion.

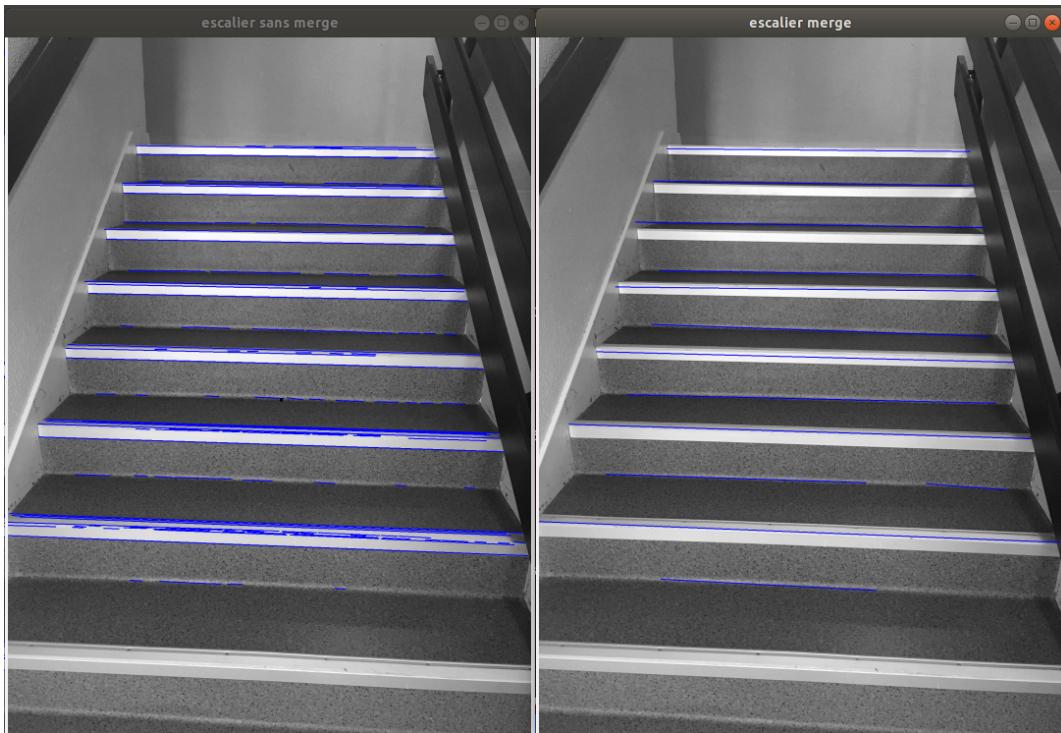


Figure 42 – Segments avant et après la fusion

Extraction du trapèze

Afin d'extraire le trapèze, nous avons procédé en plusieurs étapes. La première était d'extraire les points "droits" de tous les segments et les points "gauches" de tous les segments. Ensuite nous avons effectué une régression linéaire sur chacun de ces ensembles de points pour obtenir les coefficients des côtés du trapèze. Après nous être aperçus (et c'était logique) que les algorithmes de régression linéaires n'aimaient pas les droites verticales, nous avons effectué une régression linéaire de $x(y)$ et non pas $y(x)$.

Dans un premier temps, nous avions utilisé la fonction `sklearn.linear_model.LinearRegression` qui utilisait la méthode des moindres carrés. Cependant, cet algorithme était très sensible aux abscisses aberrantes. Nous avons donc opté pour la fonction `sklearn.svm.SVR(kernel='linear')` basé

sur l'algorithme Support Vector Regression permettait d'éliminer les points linéaires. En effet, cet algorithme prend en compte un seuil de distance autorisé entre les points pris en compte dans un régression linéaire et la droite estimée, le tout associé avec une certaine tolérance aux écarts par rapport à ce critère, nous avons donc pris soin à ne pas prendre des conditions trop strictes qui augmentaient de façon conséquente les temps de calcul et donc le retard sur la commande du drone. Les *droites limites* sont tracées de part et d'autre de la régression linéaire du bord lui-même. Le trapèze a été tracé en intersectent ces droites côtés avec les droites "haut" et "bas" des segments détectés.

Expérimentalement, on veut un faible écart autorisé, la tolérance étant faible (donc le paramètre c étant élevé). On remarque expérimentalement que $c \in [10; 100]$ convient.

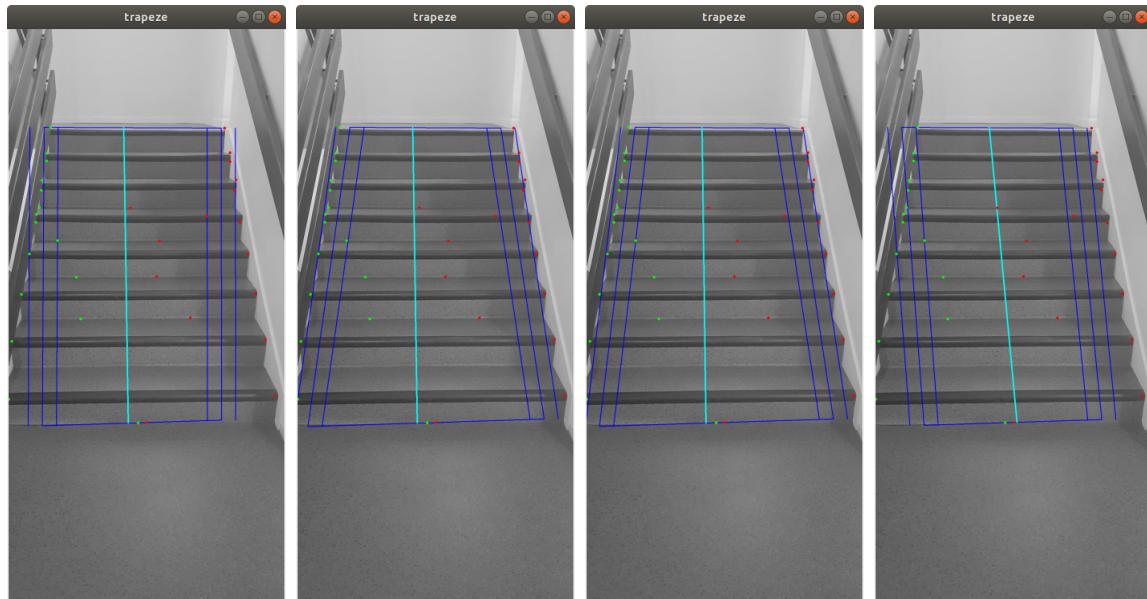


Figure 43 – Comparaison avec le trapèze tracé en utilisant la svm avec différents c (0,001, 10, 100, 1000)

On obtient les différents trapèzes suivants pour la même sortie de merge lines :

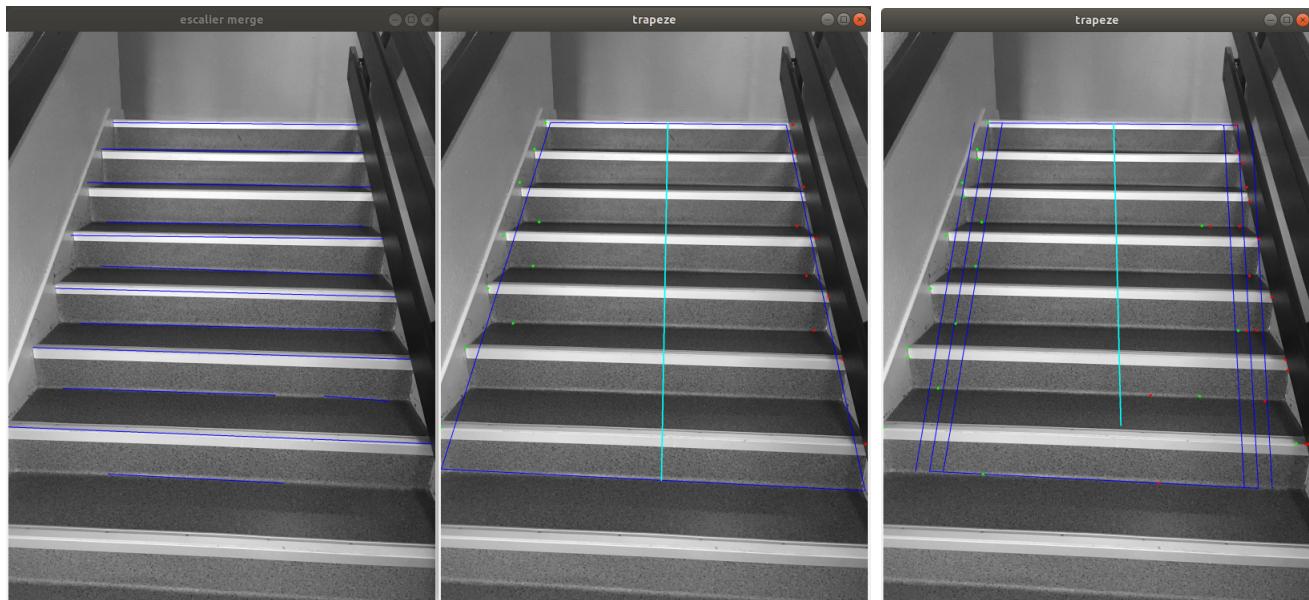


Figure 44 – Segments après merge et trapèze tracé en utilisant la méthode des moindres carrés puis la svm

Comparaison des différents estimateurs du centre des escaliers

Ces traitements étaient sensés stabiliser la détection du centre du trapèze puisqu'elles lissaient les écarts dues aux points aberrants. Cependant, après comparaison avec le centre de l'escalier déduit du centre des points composant les extrémités des marches, nous nous sommes aperçus que cette modélisation n'était pas plus stable (voir partie simulation).

3.3 Traitement d'image de la partie Détection des portes ouvertes

3.3.1 Introduction

Dans cette partie du projet, il ne s'agit pas que de naviguer dans les couloirs mais de détecter des portes. Pour effectuer cela, le drone doit effectuer des virages de plus ou moins 90 degrés lorsque le pilote en donne l'ordre puis rechercher les portes ouvertes pour pouvoir entrer. Les virages seront expliqués plus tard. Nous considérerons dans cette partie que le virage est terminé et que le drone se déplace à vitesse constante sur l'axe Y et attend qu'une porte ouverte soit détectée.// Détecter les portes ouvertes signifie que nous avons besoin d'une sorte de détection de distance utilisant le traitement d'image. Une des façons de faire est basée sur le flux optique.

3.3.2 Flux optique en quelques mots

Selon l'openCV documentation :

"Optical flow is the pattern of apparent motion of image objects between two consecutive frames caused by the movement of object or camera. It is 2D vector field where each vector is a displacement vector showing the movement of points from first frame to second."

Nous remarquons que dans cette référence, nous avons 2 algorithmes pour le flux optique, l'algorithme de Lucas-Kanade et le flux optique dense utilisant l'algorithme de Gunnar Farneback.

3.3.3 L'algorithme de Lucas-Kanade

3.3.3.1 Idée

Le premier essai consistait à utiliser le flux optique Lucas-Kanade. L'idée ici était de détecter certains points à l'aide de la fonction "goodFeaturesToTrack" fournie par openCV. Et ensuite, calculez la vitesse des points en mouvement. De cette façon, si nous découvrons une grande différence, nous dirons que nous avons une porte ouverte.

3.3.3.2 Problème

Le premier problème était que cet algorithme devait trouver des angles sur les murs et ce n'est pas toujours le cas (du moins ici à l'école, nous avons des bords et pas des coins sur les murs).

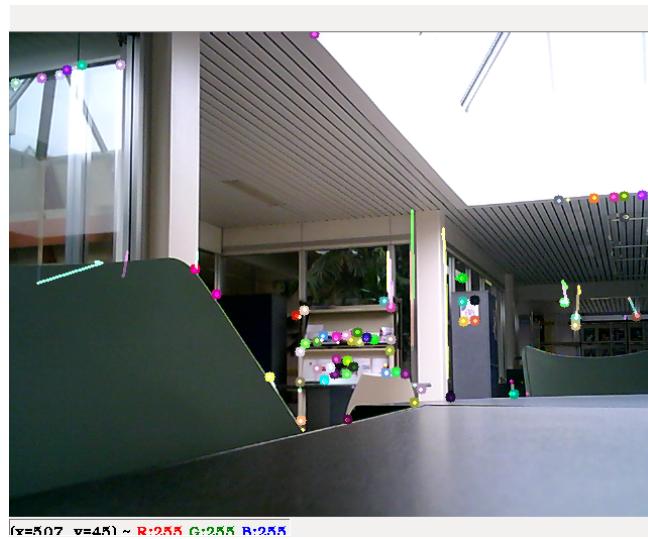


Figure 45

3.3.3.3 Solution

La solution ici est simple : il suffit de tracer une ligne horizontale sur l'image détectée et les bords se tournent vers les coins.

Cette solution peut résoudre ce problème au moins ici à l'école mais ce n'est pas une solution générale.

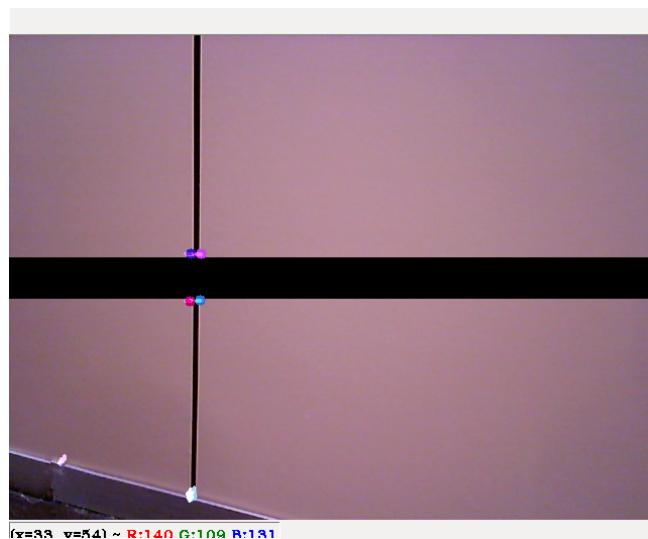


Figure 46

3.3.3.4 Problème

Un autre problème était que les informations fournies par ces points ne sont pas suffisantes, ils nous disent seulement que nous avons une porte ouverte, mais nous avons besoin de plus, nous

devons localiser cette porte. La solution ici était de remplacer cet algorithme par l'autre, le flux optique dense, qui sera expliqué dans le paragraphe suivant.

3.3.4 Flux optique dense

Le flux optique dense consiste à calculer le flux optique dans chaque pixel de l'image.

Pour appliquer cet algorithme, je vais utiliser l'algorithme Gunnar Farneback déjà implémenté dans OpenCV.

Cette fonction associe à chaque pixel de l'image un vecteur (u, v) , qui sont les valeurs du flux optique. Pour chaque valeur de X , on calcul la moyenne des $|u|$, qui est la magnitude de la composante sur x du flux, et on obtient ensuite un vecteur de valeurs qui sera utilisé pour estimer la distance dans l'étape suivante.

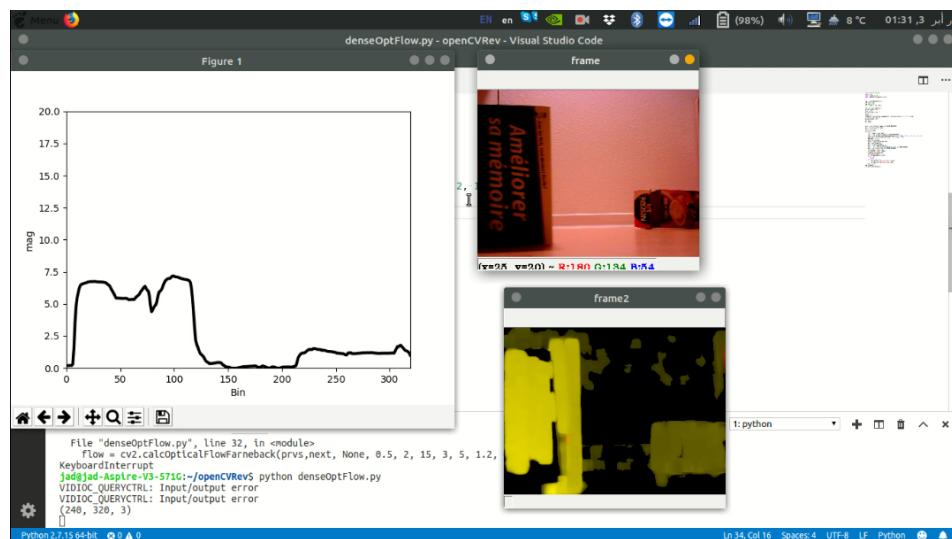


Figure 47

Dans ce projet, le drone bouge et les objets sont fixes. Ces informations nous aide à décider si nous avons une porte ouverte ou non. Dans la figure 47, On remarque que nous avons une valeur importante du flux, qui correspond au livre qui est à gauche, et une valeur presque nulle pour le milieu, et une autre valeur non nulle mais plus petite que celle d'avant, qui correspond à un objet un peu loin.

3.3.5 Estimation de la distance

Pour détecter les portes ouvertes, l'algorithme expliqué dans le chapitre précédent n'était pas suffisant.

En réalité cet algorithme calcule le flux optique d'un point de l'espace qui est une fonction de : la vitesse du drone, de la distance qui les sépare, de la largeur de l'image après redimensionnement et de l'angle alpha de la caméra. Si trois des quatre paramètres sont constants, on peut considérer que le flux optique est une fonction de la distance. Pour montrer pourquoi le flux optique ne suffit pas, expliquons comment obtenir cette formule.

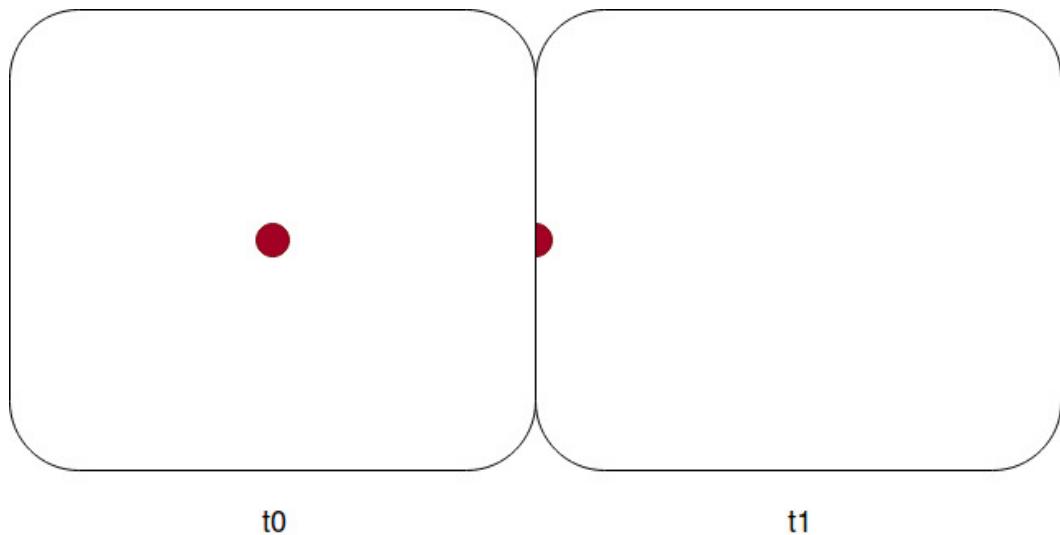
Nous allons commencer par définir quelques paramètres.

Selon Bebop_Autonomy, la vue de champ de la caméra est à 80 degrés horizontalement.

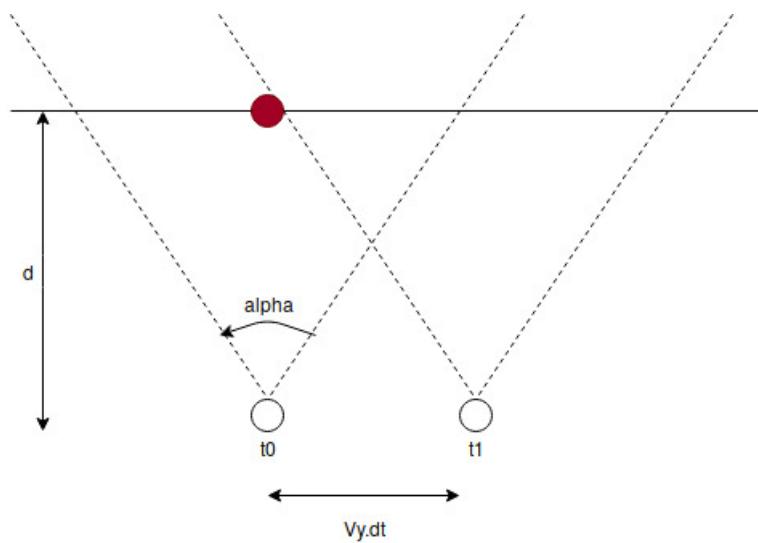
La taille de l'image utilisée dans ce projet est de (320x240).

Le drone est considéré comme se déplaçant à une vitesse constante V_y .

Considérons maintenant le cas où nous avons un point au milieu de l'écran à $t = t_0$, le drone se déplace vers la droite et le point est hors de l'écran à $t = t_1$. Ce cas est montré dans la figure suivante,



Dans la figure suivante, vous pouvez voir comment le drone se déplace entre les figures 1 et 2.



Commençons par calculer le temps dont le drone a besoin en utilisant la figure du monde réel.

$$\Delta t = \frac{\tan(\frac{\alpha}{2}) \cdot d}{V_y}$$

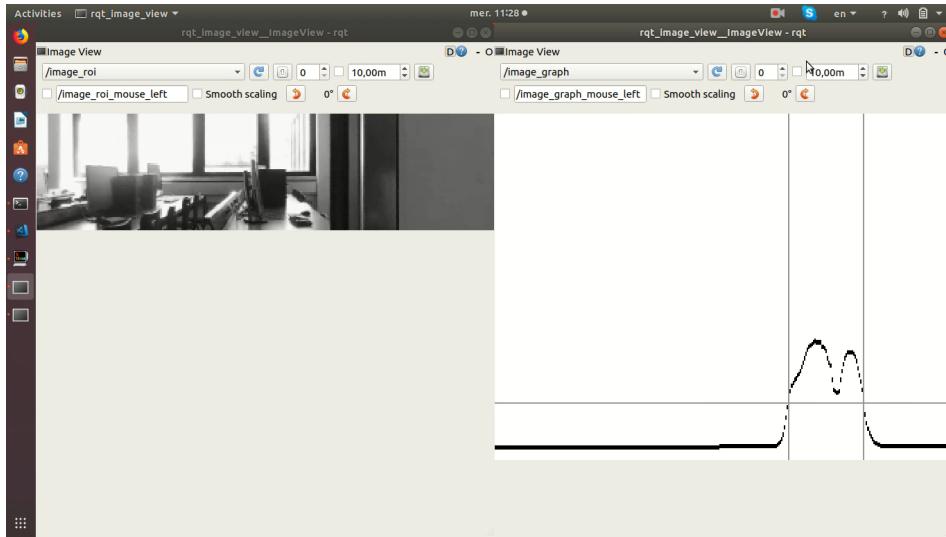
si nous calculons le même temps en utilisant ce que nous voyons sur l'image envoyée par le drone, nous aurons :

$$\Delta t = \frac{\frac{W}{2}}{\|f\|}$$

comme les deux Δt sont égaux, nous pouvons obtenir la relation suivante :

$$\|f\| = \frac{\frac{W}{2}}{\tan(\alpha)} \frac{V_y}{d}$$

3.3.5.1 Test dans l'Ecole

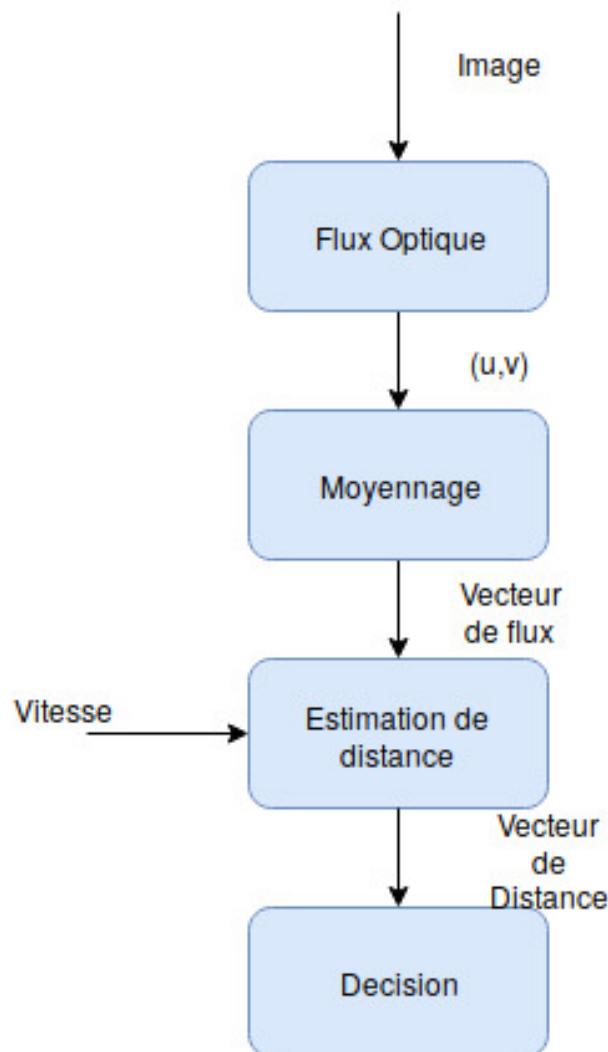


3.3.6 Décision

Le fait d'avoir un flux optique très faible ne signifie pas que l'objet est loin, il y a une grande possibilité que ça soit un mur de couleur uniforme, comme dans la figure précédente par exemple. Alors la porte ouverte n'est pas seulement une région large où le flux est faible il faut avoir une autre condition. Mais avant discuter cette condition, il faut, en se basant sur l'information récupérée, diviser l'image en régions où la distance est plus grande qu'une certaine valeur et d'autres régions où la distance est plus petite que cette valeur. On prend ensuite les régions suffisamment larges et qui correspondent à une grande distance et on prend aléatoirement des colonnes et on calcule la variance des pixels dans ces colonnes, qui nous donne une information sur la variabilité des couleurs, si cette variation est grande alors il s'agit d'une porte ouverte. Sinon c'est un mur.

Donc la porte ouverte est une région avec un flux optique faible et une grande variation de couleurs.

3.3.7 L'algorithme en une seul image



4 Commande du drone avec ROS

4.1 Les modules pour la commande avec vitesse constante

Dans la simulation, la commande sur l'axe Y est une commande de vitesse, ce qui signifie que l'envoi d'une commande de 0.2 sur l'axe Y par exemple, donnera un twist en Y qui fera bouger le drone avec une vitesse de 0.2 sur cet axe. Cependant ce n'est pas le cas dans le cas réel. Selon la documentation sur l'autonomie du bebop, les twists linéaires en x et y ne sont pas des vitesses en réalité, mais respectivement des angles de tangage et de roulis. Cela signifie que c'est une commande d'accélération et non une commande en vitesse. Pour faire face à cela, la solution a été de créer un ensemble de noeuds permettant d'appliquer le schéma suivant :

1. Lire la vitesse du drone dans le topic /bebop/odom.
2. Calculer la différence entre la vitesse souhaitée du drone et celle réelle.
3. Calculez le twist sur Y en utilisant la formule suivante :

$$newComm = K_p * Err + K_i * SumErr + K_d * DiffErr$$

Avec Err la différence entre la vitesse cible et celle mesuré,

$SumErr$ la somme de toutes les erreurs précédentes,

et $DiffErr$ la différence entre l'erreur actuelle et l'erreur précédente.

Les K_p, K_i, K_d sont des constantes définies de manière empirique.

4.1.1 Résultats :

Avant de commencer à présenter les résultats, il convient de mentionner que, selon la documentation sur l'autonomie de Bebop, la fréquence de ce sujet est de 5 Hz.

Tous les tests effectués ici sont réalisés avec une valeur cible de 0,8 sauf indication contraire

Le premier test portait sur le contrôle P pur, ce qui signifie que les K_i, K_d sont mis à zéro. Nous obtenons la courbe suivante :

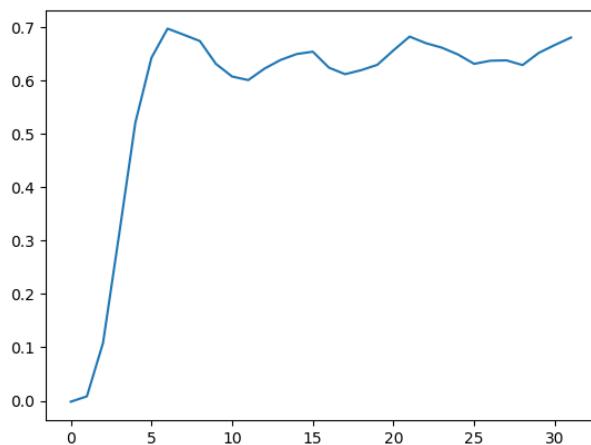


Figure 48 – Vitesse du drone

Nous pouvons constater qu'après presque une seconde, la valeur de la vitesse est d'environ 0,65, ce qui signifie la présence d'une erreur permanente dans ce contrôle. Le second test était pour le contrôle PI, ce qui signifie que K_d est défini sur zéro. Une saturation a été ajoutée sur le I, ce qui signifie que la somme n'est calculée que si la valeur de la commande est inférieure à une valeur spécifique. Nous obtenons la courbe suivante :

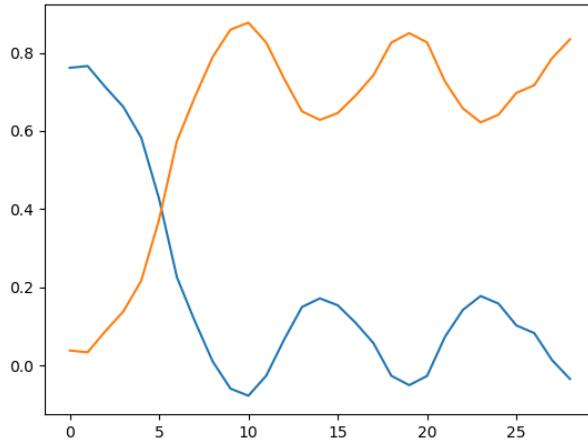


Figure 49 – Vitesse et erreur

Nous pouvons voir ici que les résultats sont un peu meilleurs puisque les valeurs sont autour de 0,8 mais que nous avons des oscillations très importantes. Le troisième test utilisait le contrôle PID. Nous obtenons les courbes suivantes :

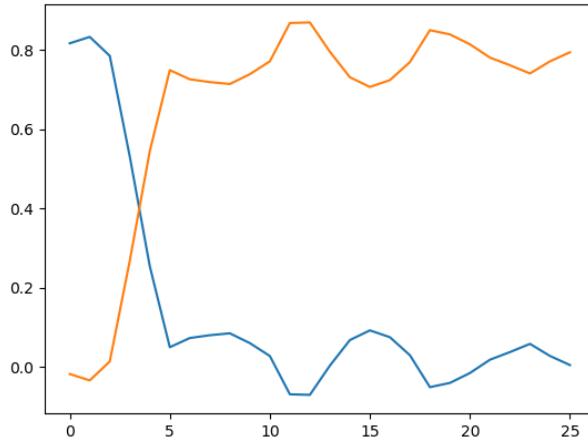


Figure 50 – Contrôle PID

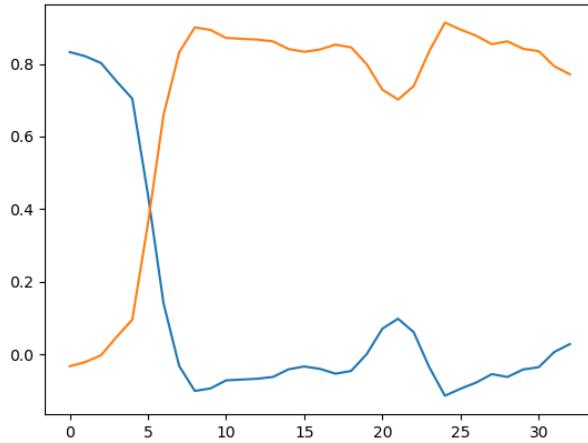


Figure 51 – Contrôle PID

Nous pouvons voir que les résultats sont très bons dans les deux figures et que les problèmes rencontrés dans les deux figures sont résolus.

Le temps de réponse pourrait être amélioré en ajustant les paramètres. Et nous avons ici un autre test pour un objectif de 0.5 :

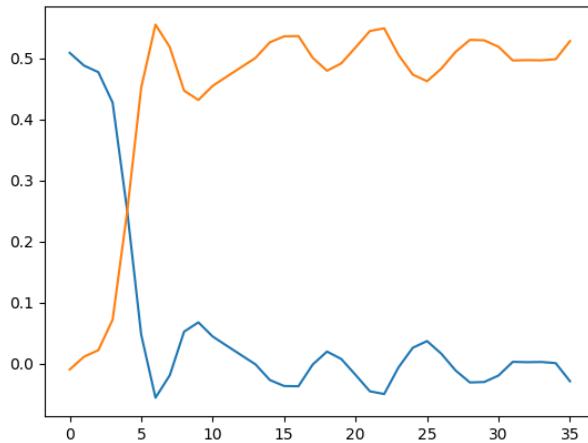


Figure 52 – Contrôle PID

L'utilisation de ce module est expliquée en détails dans le README sur le Gitlab du projet.

4.2 Commande pour tourner d'un angle donné

Les modules expliquées au-dessus sont utilisées pour créer un autre module qui nous permet de faire tourner le drone d'un angle précis.

4.2.1 Les Quaternions

Pour pouvoir faire ça il nous faut lire la rotation du drone qui est fournie sous forme d'un quaternion reçu dans le message d'odométrie. Le message Odometry reçu du drone est en fait composé de 2 parties, le "Twist" et la "Pose". Dans ce paragraphe, nous parlerons de la pose car nous devons obtenir l'angle de lacet du drone.

La "Pose" contient également 2 champs, la position et l'orientation.

L'orientation est représentée sous la forme d'un quaternion.

Dans notre cas, nous devons convertir un quaternion en angle pour obtenir l'angle autour de l'axe Z souhaité. Selon Wikipedia, la formule est la suivante :

Considérons que nous avons le vecteur Quaternion suivant : $q = [q_w q_x q_y q_z]^T$

On peut associer un quaternion à une rotation autour d'un axe par l'expression suivante :

$$q_w = \cos\left(\frac{\alpha}{2}\right)$$

$$q_x = \sin\left(\frac{\alpha}{2}\right) * \cos(\beta_x)$$

$$q_y = \sin\left(\frac{\alpha}{2}\right) * \cos(\beta_y)$$

$$q_z = \sin\left(\frac{\alpha}{2}\right) * \cos(\beta_z)$$

Où α est l'angle de rotation et $\cos(\beta_x)$, $\cos(\beta_y)$ et $\cos(\beta_z)$ sont les "cosinus de direction" localisant l'axe de rotation.

Si v est un vecteur euclidien unitaire dans un espace euclidien tridimensionnel, $v = v_x.e_x + v_y.e_y + v_z.e_z$, alors $\cos(\beta_x) = v_x$

4.2.2 Calcule de commande

Après avoir lu le quaternion, et définit l'angle cible, on peut calculer l'angle par la simple formule $vit = K * (cible - angle)$.

Ensuite on passe cette valeur au module expliqué précédemment et il s'occupe du reste.

4.3 Commande pour naviguer dans les couloirs

4.3.1 Introduction

Dans cette section, nous parlerons du contrôle du drone et de la manière dont les lignes détectées, les points de fuite et autres informations sont utilisés afin de contrôler la navigation du drone dans le couloir. Nous suivrons le schéma suivant pour élaborer la commande.

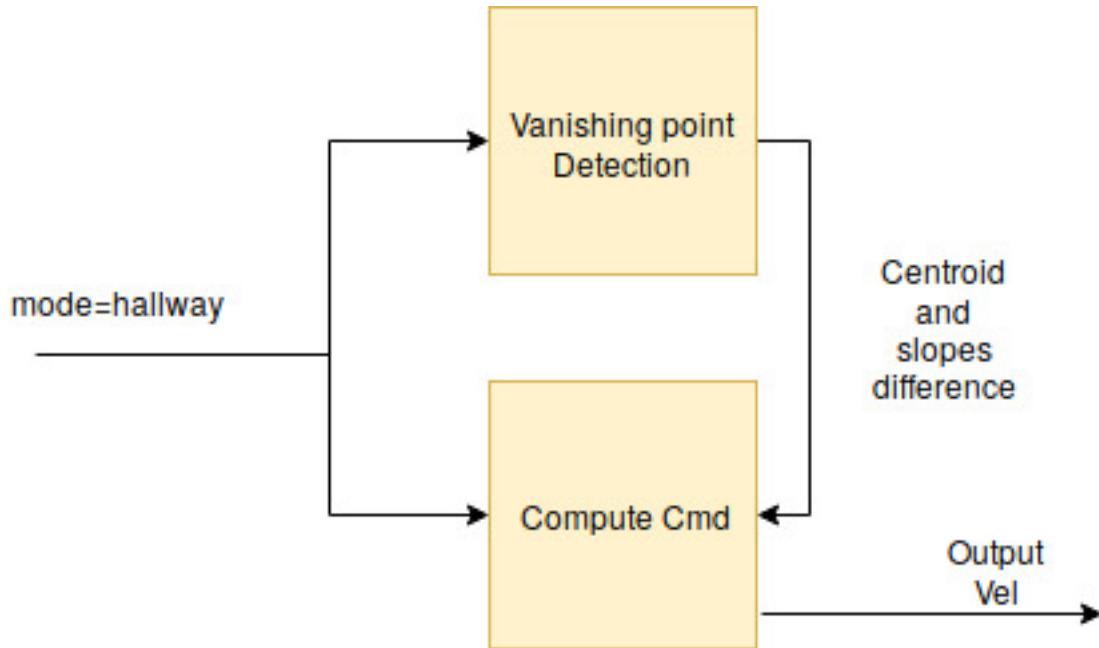


Figure 53 – Commande pour les couloirs

4.3.2 Aller droit

Le premier essai consistait simplement à faire correspondre le point de fuite et milieu de l'écran en commandant l'angle de lacet du drone.

Dans cette section, nous avons utilisé un contrôleur P.

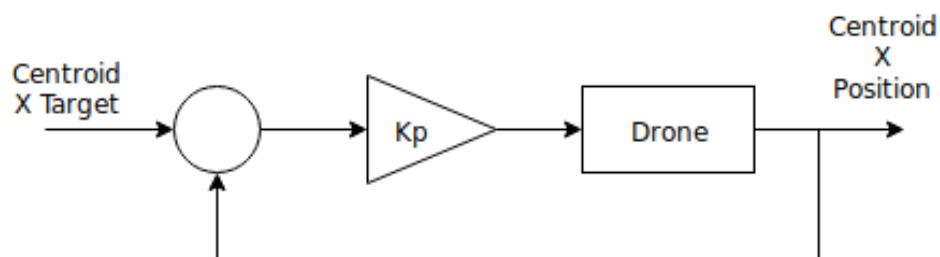


Figure 54 – Contrôleur P

En utilisant ce contrôle, le drone a pu se déplacer droit dans le couloir mais pas au milieu du couloir. Être près du mur avec un vrai drone est très risqué, car il n'est jamais stable ni figé dans l'espace

en raison de la turbulence du vent causée par les pales en rotation.
 Dans le paragraphe suivant, nous parlerons de la solution à ce problème.

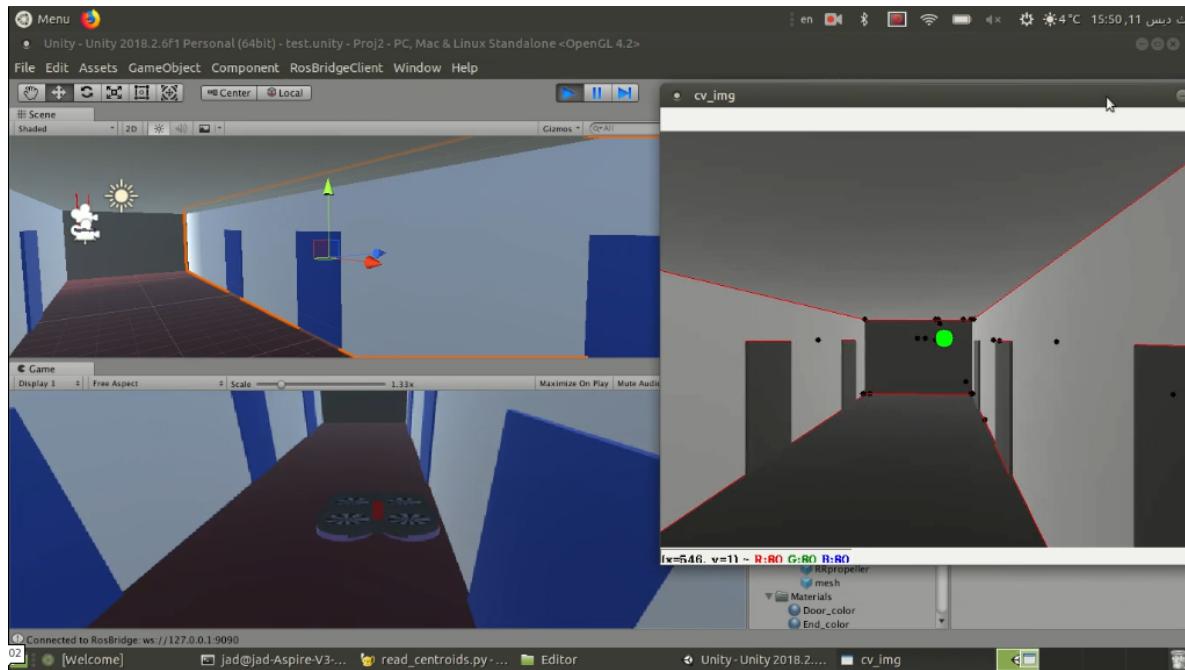


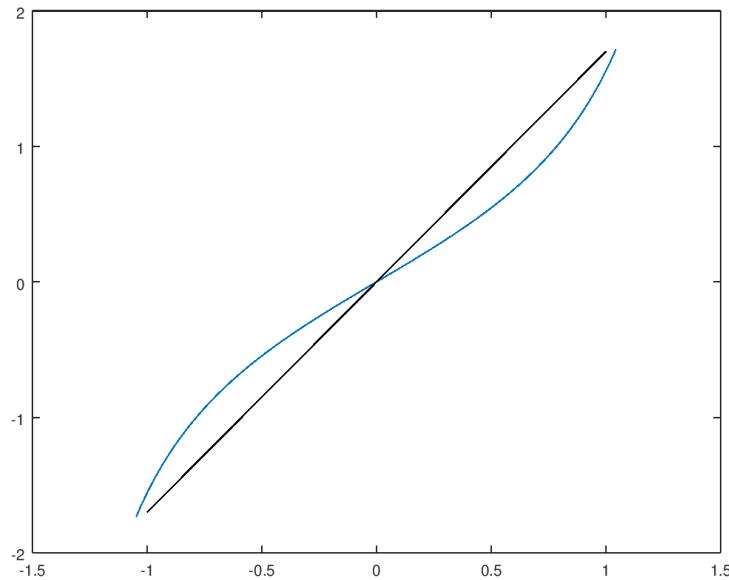
Figure 55 – Test de commande

4.3.3 Se déplacer au centre du couloir

Pour laisser le drone bouger au centre du couloir, nous devons avoir une idée des images concernant la position du drone dans le couloir, en particulier la distance entre les drones et les murs à gauche et à droite.

Pour ce faire, nous utilisons la différence entre les pentes déjà calculées.

Vous pouvez remarquer ici l'utilisation de la pente qui est la tangente de l'angle entre l'axe des x et la ligne, cela signifie que ce n'est pas linéaire, ce qui peut poser un problème de proportionnalité puisque nous utilisons un contrôleur P, mais comme vous pouvez le voir dans la figure suivante



La tangente de x pour x entre $-\frac{\pi}{3}$ et $\frac{\pi}{3}$ est assez proche d'une ligne.

Ce contrôle peut être modifié ultérieurement si nous voyons que cette approximation est insuffisante et le remplace par un contrôle sur le vecteur de direction de la ligne par exemple.

Pour commander la vitesse sur Y , et comme déjà mentionné, nous avons utilisé la différence entre les pentes et cette valeur multiplié par une constante était la vitesse cible, et ensuite les modules expliqués au début du chapitre s'occupent du reste.

4.4 Commande pour le passage d'escaliers

4.4.1 Introduction

Le mode *stairs* (escaliers) est constitué de deux phases : la première, *go_to_stairs_center*, qui consiste à localiser l'escalier, se diriger vers lui de sorte à fixer sans cesse son centre pour ne pas le perdre de vue, et s'arrêter une fois en face. Elle envoie ensuite un signal de fin de tâche afin que commence la seconde phase. Commence alors la phase *get_up_stairs*, qui consiste à monter les escaliers. Cette seconde phase ne publie aucun message de fin de tâche. Nous expliquerons plus tard pourquoi. On a donc l'architecture ROS suivante.

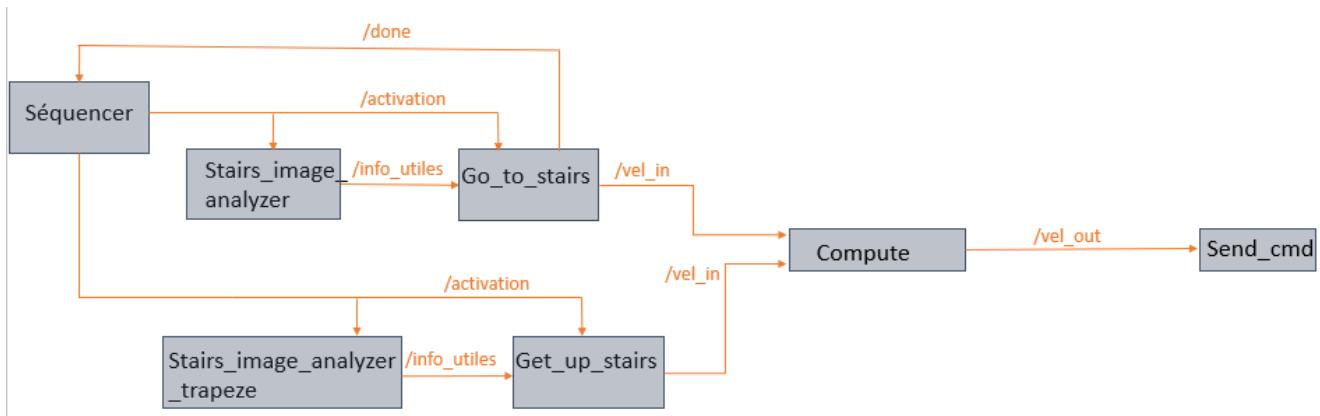


Figure 56 – Schéma récapitulant l'architecture ROS pour la montée d'escalier

On remarque que cette architecture comprend deux *image processors* dédiés à chacune des phases de ce mode. En effet, on pourrait croire que si l'on peut détecter un escalier dans une mauvaise configuration (le cas favorable étant lorsqu'on en est proche et de face) alors on est aussi capable de le détecter dans une bonne configuration, c'est le cas de la phase *get_up_stairs*. Nous expliquerons dans les parties suivantes la nécessité de cette disjonction.

Dans tous les cas, si l'*image processor* ne détecte aucun escalier dans son champ de vision, le drone continue dans la même direction qu'à l'instant précédent, espérant voir ledit escalier en se déplaçant. Si le drone ne voit aucun escalier au moment de l'activation du mode, il restera à l'arrêt. Dans un cas favorable, sa vitesse de dérive lui permettra de se déplacer suffisamment pour voir un escalier apparaître. Si tel n'est pas le cas, l'utilisateur comprendra aisément qu'il faudra changer de mode afin de se placer dans une disposition qui permette au drone de voir un escalier. De même si le drone confond une autre entité avec un escalier et se dirige vers elle.

4.4.2 Phase de placement

Cette phase consiste à détecter l'escalier et à se diriger vers celui-ci. Elle se découpe en 3 parties :

- Localiser les escaliers (et attendre en cas d'absence de détection)
- Se diriger vers les escaliers
- Un fois positionné correctement par rapport aux escaliers, s'arrêter et envoyer un message de fin de tâche

4.4.2.1 Stratégie

Lorsque l'utilisateur met en place le mode escalier, il s'agit dans une première phase de détecter les escaliers dans le paysage et de se positionner correctement de sorte à pouvoir dans une deuxième phase monter les escaliers.

Ici, on doit se placer dans un cas général puisque l'on ne se trouve pas nécessairement proche de l'escalier. Les lignes de l'escalier ne prennent pas nécessairement la majeure partie de l'image détectée, ne sont pas nécessairement très nombreuses, et peuvent avoir un angle important par rapport à l'horizontale. On a montré précédemment comment détecter un escalier même dans ces conditions non favorables.

Maintenant, comment se diriger vers lui ? Faut-il, comme dans les couloirs, viser son "point de fuite" ?

On sait tout d'abord que le point de fuite des marches donne une indication de l'angle auquel on se trouve de l'escalier, et que les escaliers se trouvent dans la direction "orthogonale", cependant, quid de la divergence de ce point lorsque les escaliers sont de face (cas favorable) ?

On peut alors considérer les marches comme globalement parallèles sur l'image, tracer ainsi le trapèze formé par ses côtés et viser le point de fuite de ce trapèze ou son milieu.

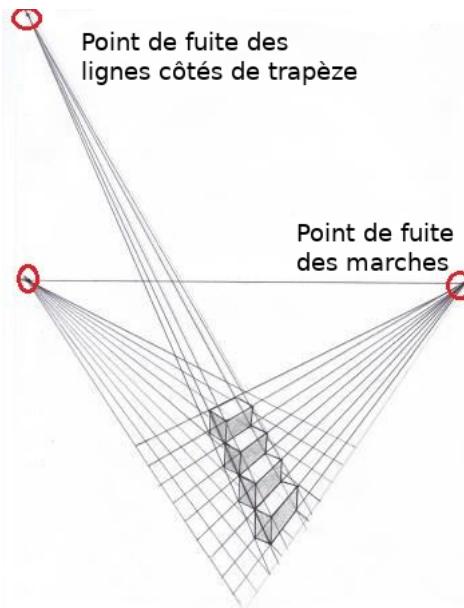


Figure 57 – Point de fuite des marches et des côtés du trapèze

Cependant, dans une phase d'approche, ce trapèze est trop mal dessiné pour pouvoir le prendre comme référence (on bonne partie de l'escalier peut par exemple être masquée par un mur), et en phase de montée, ce point risque également de diverger.

On pense alors tout naturellement à une solution simple qui est plutôt la suivante : viser le milieu de l'escalier. Celui-ci est facile à calculer (même s'il peut être estimé de plusieurs manières plus ou moins complexes). En mettant en œuvre cette technique, nous nous sommes rapidement aperçus d'un problème : dans le cas où un obstacle non complètement opaque (rambarde, mur transparent) se trouve aux abords de l'escalier on fonce droit vers l'obstacle (Voir figure 35 de la partie traitement

d'image) ! On veut donc dans un premier temps viser "l'entrée" de l'escalier, à savoir ses premières marches. Dans cette première phase, on réduira donc l'escalier à ses quatre marches les plus basses dont on estimera le centre (estimateur choisi : moyenne des milieux des segments) pour le viser (voir partie traitement d'image et les figures 38 et 39). Maintenant que l'on a choisi la cible, il s'agit de choisir la trajectoire à imposer au drone pour atteindre l'endroit choisi comme adapté à une montée d'escalier. Le "cas défavorable" de base étant le suivant : on se trouve dans un couloir à l'orthogonale duquel se trouve un escalier. Comme on veut toujours garder l'escalier dans le champ de vision (d'une part pour savoir le localiser en permanence, d'autre part pour pouvoir s'adapter à n'importe quelle configuration) on va donc se tourner (en z) vers l'escalier. On continuera sans cesse d'avancer au cours du processus pour deux raisons : d'une part garder l'escalier toujours dans le champ de vision, d'autre part pour stabiliser le signal est réduire la sensibilité aux perturbations (en effet, si on détecte un mauvais élément sur l'image, on peut espérer qu'à l'instant d'après on détectera à nouveau l'escalier). Comme le signal est sans cesse corrigé et qu'on avance en même temps qu'on tourne, on obtiendra une trajectoire globalement courbe jusqu'aux escaliers.

4.4.2.2 Localisation de l'escalier

La localisation de l'escalier, où plus précisément la détection du milieu des bas des marches de l'escalier se fait avec le traitement d'image (voir section1). Nous avons écrit un noeud, nommé *image processor* qui, une fois activé, traite l'image. Il en déduit 3 choses :

- s'il y a un escalier
- s'il y en a un, la différence entre le centre du bas des marches et le milieu de l'image (en abscisses et en ordonnées)
- l'angle des segments de l'escalier.

Ces 4 informations sont envoyées sur 4 topics différents. Nous en déduirons par la suite la commande du drone.

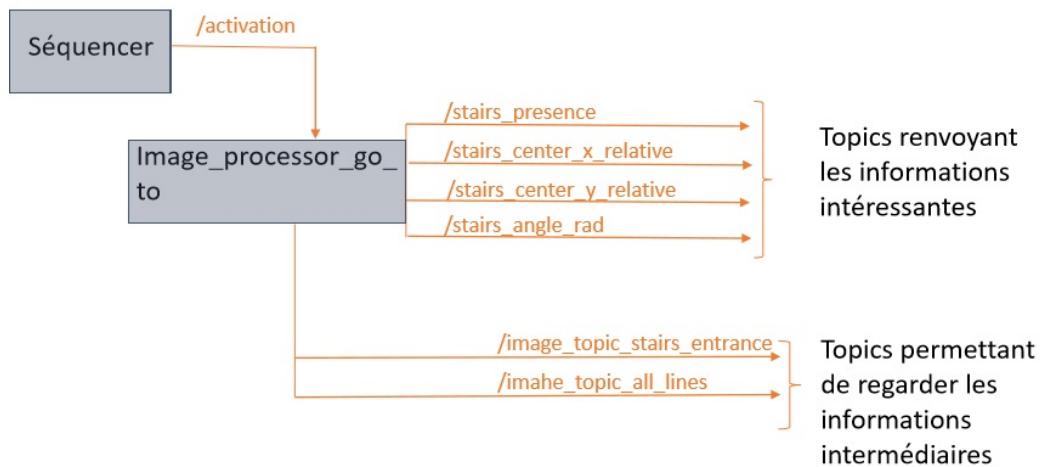


Figure 58 – Schéma de fonctionnement du noeud Image Processor go to

4.4.2.3 Élaboration de la commande

Afin d'élaborer la commande nous allons, comparer le target point (le x de la ligne verticale rouge de la figure 64) avec le milieu de l'image détecter par le drone (ligne verticale noire sur la figure 64). Grâce à la différence entre ces 2 points nous allons lui dire de tourner, à droite ou à gauche en fonction de cette différence. Le premier problème que nous avons rencontré est que, pour rejoindre l'escalier, nous ne pouvions pas vraiment nous déplacer sur y car sinon le drone peut rentrer dans un mur sans le voir. De plus nous avons pensé à commander le drone afin qu'il fasse un arc-de-cercle pour rejoindre l'escalier. Cependant, n'ayant aucune information sur la distance avec l'escalier cette idée à vite été écartée. Nous avons décidé que nous ne commanderons le drone que sur x (c'est à dire vers l'avant et l'arrière) et sur l'angle z (c'est à dire pour tourner à droite ou à gauche). Ainsi le drone, avance et se tourne sans cesse pour fixer le milieu de l'escalier.

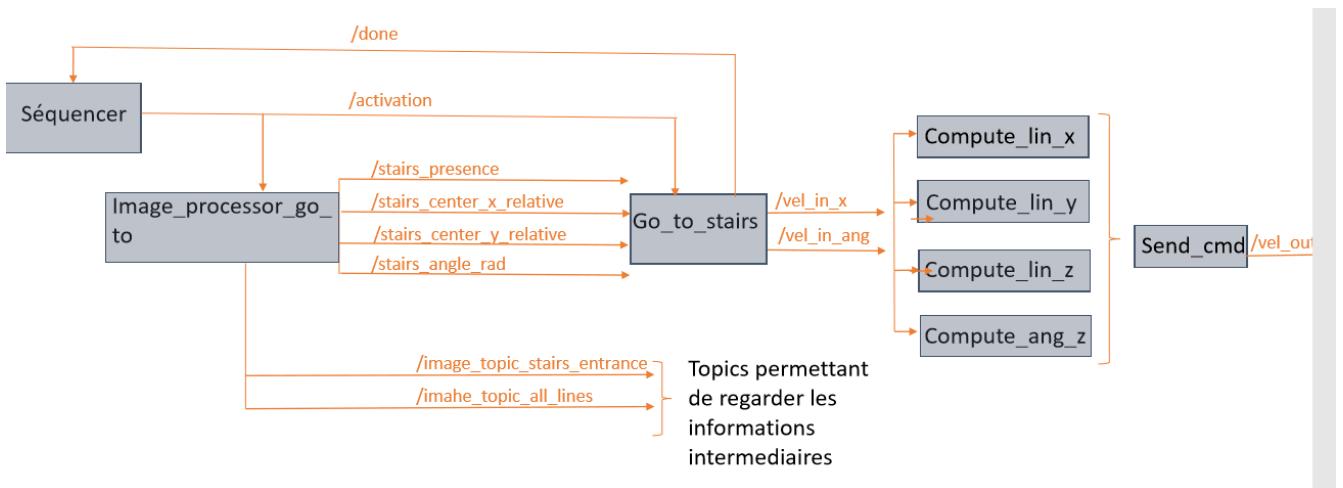


Figure 59 – Schéma du fonctionnement de la phase où le drone va vers l'escalier

Sur la figure suivante :

- Les lignes en bleues sont les lignes de l'escalier détectées et fusionnées.
- Les lignes vertes sont les lignes du "bas" de l'escalier.
- La ligne rouge horizontale correspond au y moyen des lignes du bas de l'escalier
- La ligne noire horizontale détermine la condition pour terminer la phase "go to stairs". C'est à dire lorsque le y moyen des lignes du bas de l'escalier (ligne rouge) passe en dessous de cette ligne, la phase se termine.
- La ligne noire verticale est la ligne verticale passant par le milieu de l'image
- La ligne rouge verticale correspond au x moyen des segments du bas de l'escalier

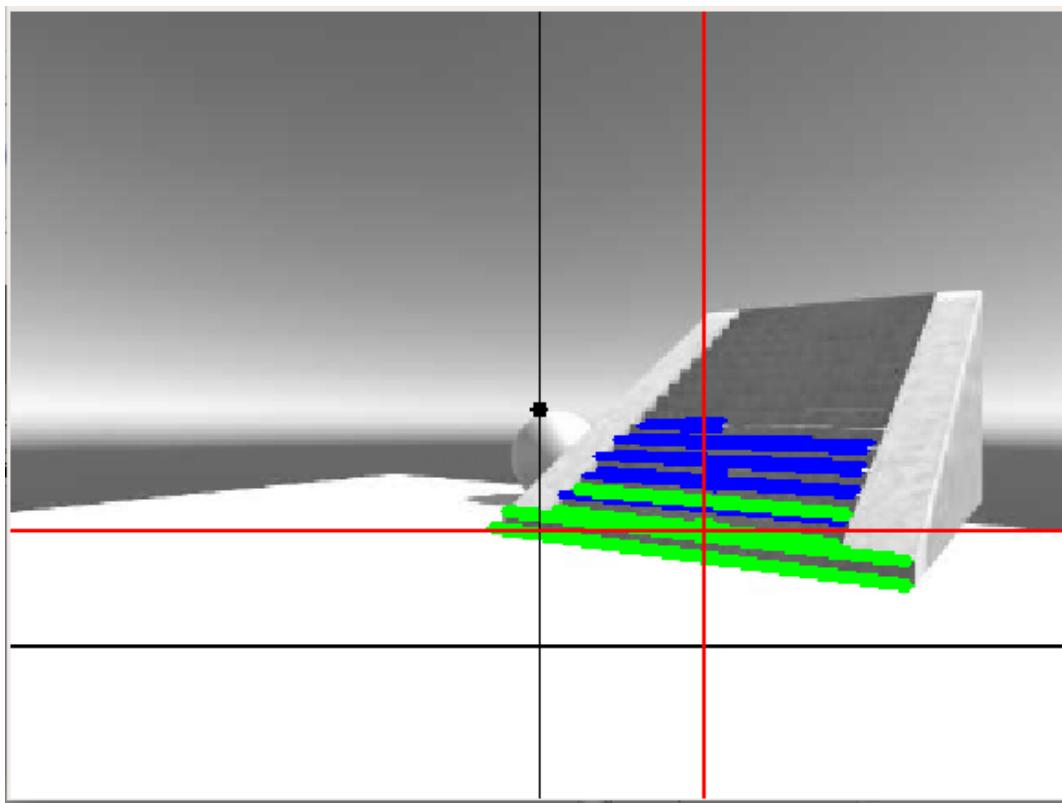


Figure 60 – Photo de ce que voit le drone en simulation

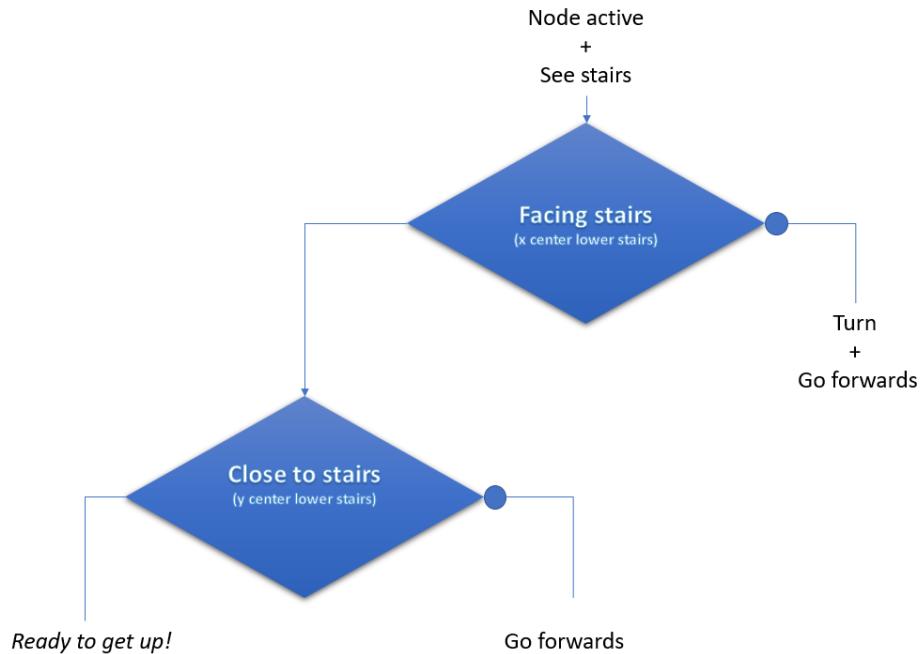


Figure 61 – Schéma logique du callback de la phase où le drone va vers l'escalier

Une fois la commande en vitesse déterminée, il faut la convertir en commande compréhensible par le drone. Voir partie 4.1

4.4.2.4 Fin de tâche

On s'arrête lorsque l'on est suffisamment proche de l'escalier pour que la ligne la plus basse de l'escalier se trouve dans le quart le plus bas de l'image. Le drone s'arrête et le message de fin de tâche est envoyé au séquenceur par le nœud de commande. On considère comme bas de l'escalier non pas la marche la plus basse car des traits aux sol peuvent être considérés comme des marches de l'escalier mais plutôt la moyenne des y des 10 marches les plus basses, ce qui diminue donc la sensibilité de la mesure aux erreurs de détection.

4.4.2.5 Problèmes rencontrés et solutions

Le premier problème rencontré et que pour un escalier de type 1, c'est à dire un escalier avec des balustrades de sécurité transparentes, le centre de l'escalier n'était pas au bon endroit et le drone rentrait dans la barrière. Nous avons donc opté pour la solution de détection du bas de l'escalier ce qui fonctionne relativement bien.

4.4.3 Phase de montée

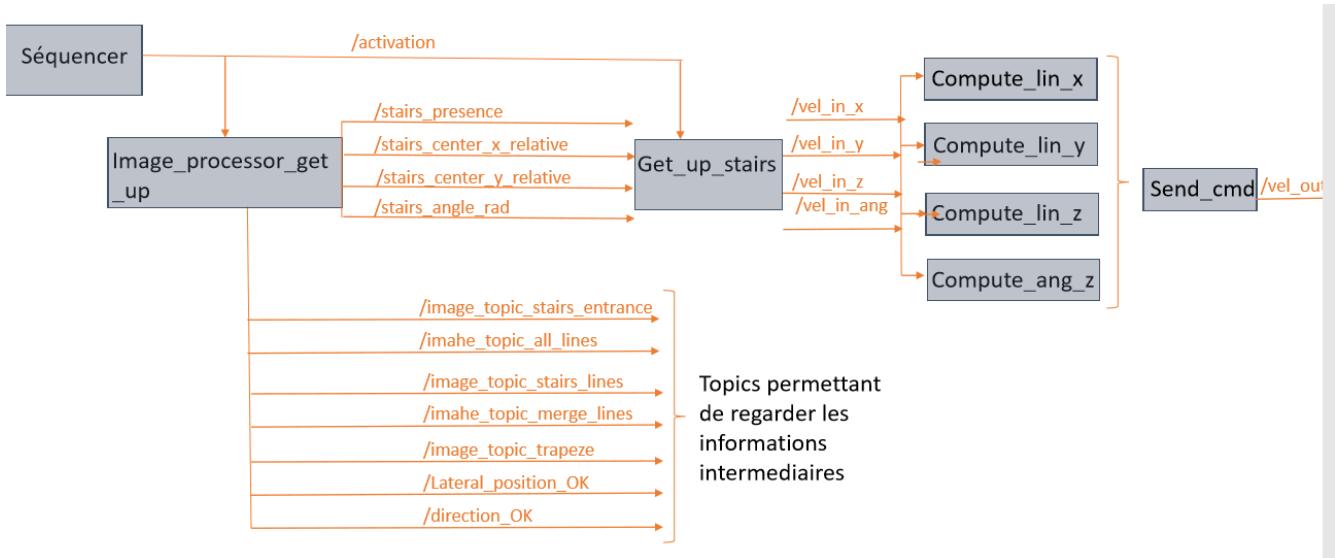


Figure 62 – Schéma montrant le fonctionnement de la phase get-up

4.4.3.1 Stratégie générale

Nous entrons dans cette phase lorsque le drone est proche de l'escalier.

En termes de positionnement, on peut se retrouver face aux problématiques suivantes :

- **orientation** : Comme la phase précédente n'assure pas nécessairement la bonne orientation du drone par rapport aux escaliers nous avons fait une phase transitoire au début de la phase get up. On utilise l'angle des escaliers afin d'orienter le drone. Si l'angle des escaliers est comme sur la figure 63 on lui demande de tourner vers la droite et vice versa. On a

cru un instant stabiliser encore plus le centre de l'escalier en prenant comme estimateur le centre du trapèze mais une étude comparative a montré que ce n'était pas le cas, même on supprimant les points aberrants de la régression linéaire des deux côtés, et nous avons donc conservé comme référence le repère le plus stable, à savoir le centre des extrémités des segments fusionnés de l'escalier.

- **centrage** : Par ailleurs, la phase précédente n'assure pas nécessairement d'être centré vis à vis de l'escalier, il nous faut donc détecter à nouveau le centre de l'escalier. Nous nous sommes rapidement aperçus que le signal fourni par les marches les plus basses de l'escalier est beaucoup trop instable en phase de montée puisque les marches du bas disparaissent au fur et à mesure qu'on monte les escaliers, ce qui crée soudainement un gros écart dans le positionnement dans le centre de l'escalier. Dans cette phase, on prend donc en compte "toutes" les marches (détectées) de l'escalier afin de diminuer la sensibilité aux imperfections de la détection. Ainsi on utilise la différence entre les x du milieu de l'escalier et celui du milieu de l'image du drone afin de le commander en y. Si la ligne verticale rouge (celle du milieu de l'escalier) est à droite de la ligne verticale noire (celle du milieu de l'image), on lui demande de se déplacer vers la gauche (on lui envoie un y-cmd positif) et vice versa.

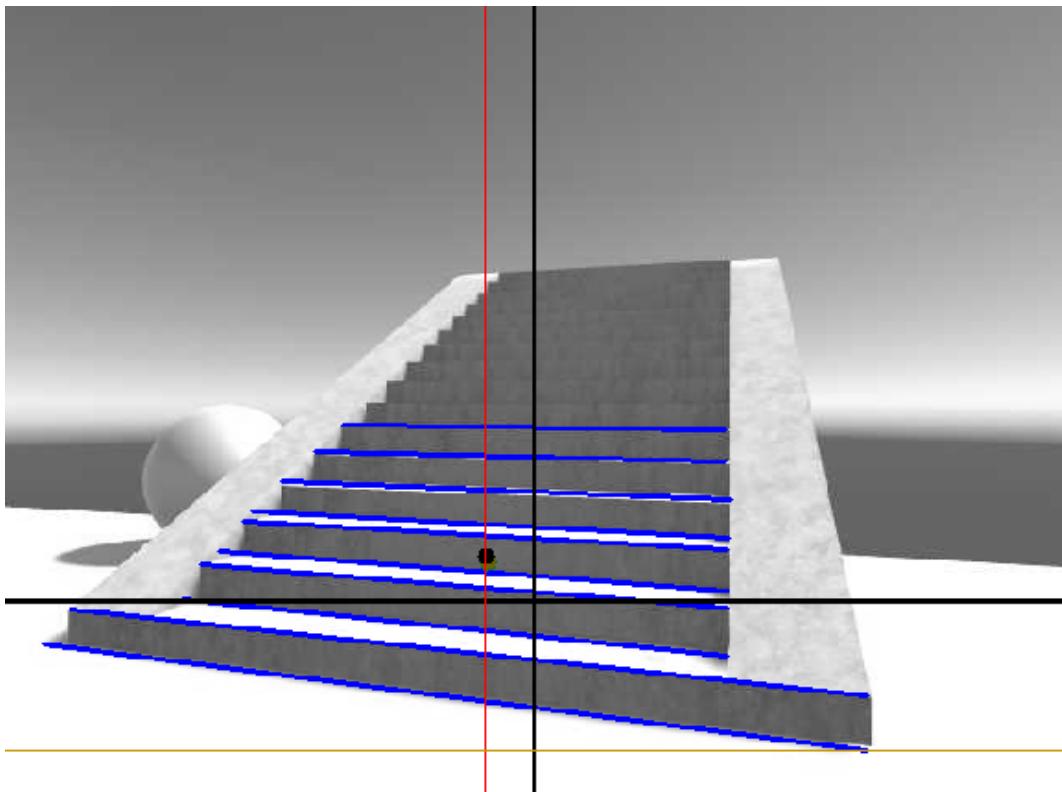


Figure 63 – Ce que voit le drone lorsqu'il n'est pas encore centré ni bien orienté

- **altitude** : Plus de souci de détection de lignes du sol. On repère donc le bas de l'escalier par la marche la plus basse et on impose que le drone monte quand cette marche est au-dessus du tiers de l'image (ligne noire de la figure 64). On peut voir que sur cette figure, la ligne la plus basse de l'escalier détectée est bien au-dessus de la ligne noire. On peut à première vue, trouver cette commande illogique. Cependant du fait de la focale du drone, lorsqu'il est trop proche de l'escalier, il ne détecte pas les lignes du bas de l'escalier mais que celle du milieu de celui-ci. Ainsi quand le y-max passe au-dessus de cette ligne, cela signifie que le drone est

trop proche de l'escalier et qu'il doit monter.

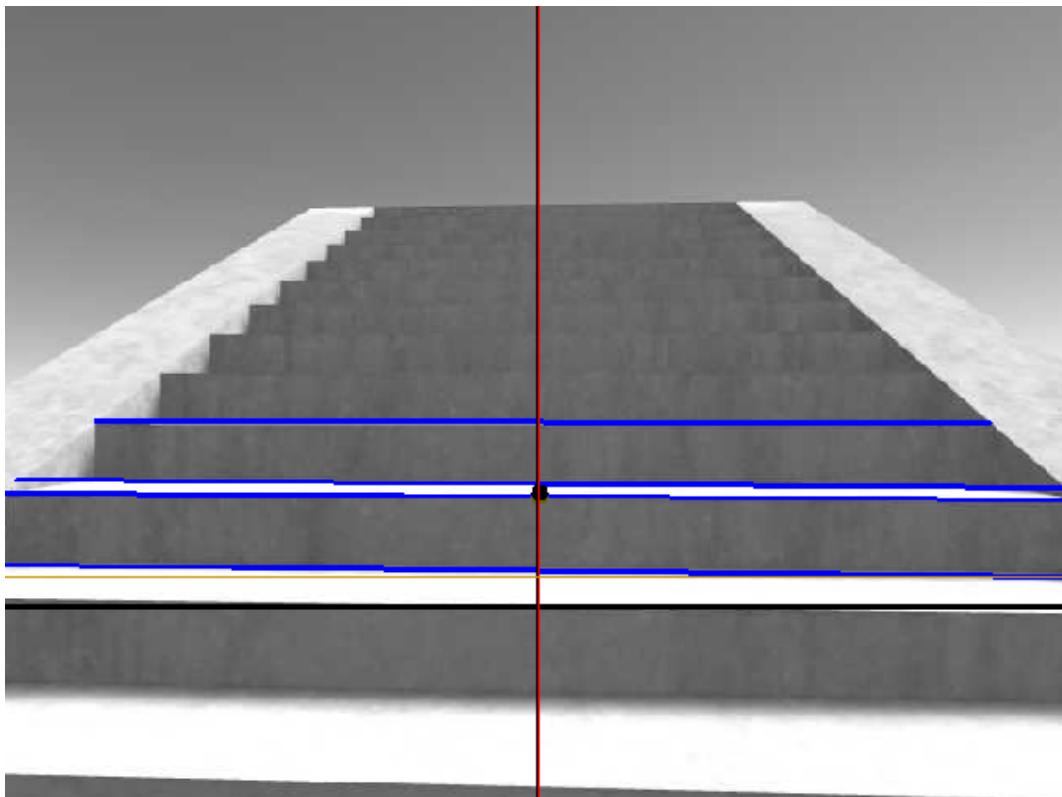


Figure 64 – Photo de ce que voit le drone lorsqu'il doit monter

On peut remarquer sur cette dernière photo que le drone est parfaitement centré et bien orienté. Pour stabiliser le signal et diminuer la sensibilité aux perturbations pendant la montée, on donnera sans cesse la consigne d'avancer.

4.4.3.2 Élaboration de la commande

Comme expliqué précédemment cette phase comporte deux parties.

La première consiste à bien se repositionner. On oriente le drone grâce à l'angle de l'escalier. En parallèle, afin de pas perdre de vue l'escalier, on le déplace horizontalement, commande établie grâce à la différence des x de l'image entre le milieu de l'image et le milieu de l'escalier.

Une fois le drone bien orienté et bien en face de l'escalier. Il peut enfin avancer et monter l'escalier.

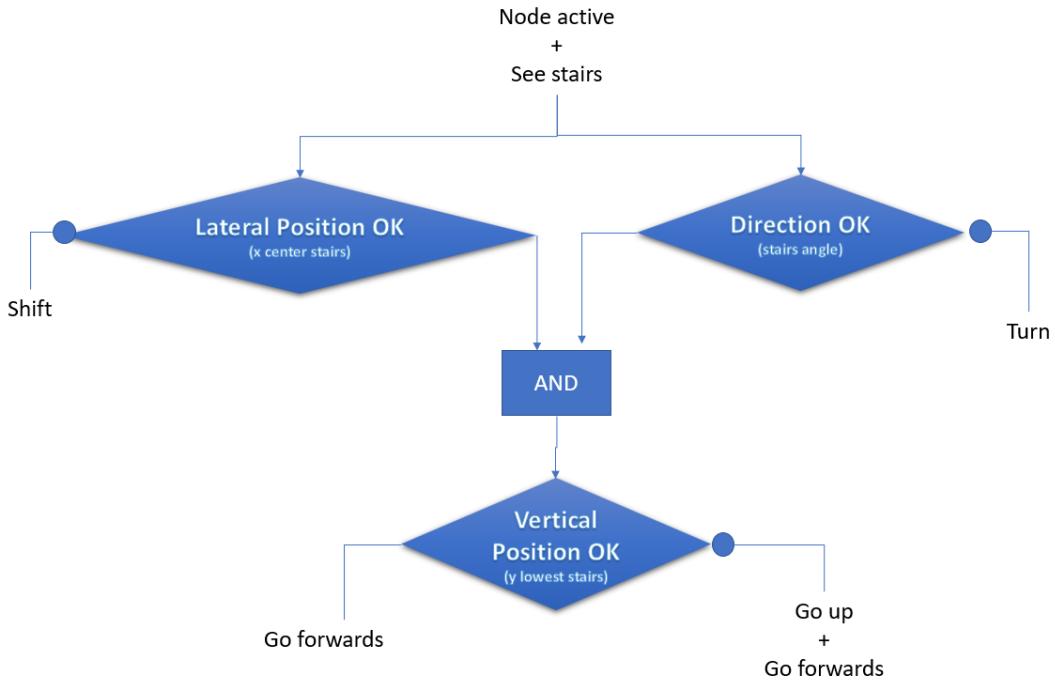


Figure 65 – Schéma logique du callback de la phase où le drone monte l'escalier

4.4.3.3 Fin de tâche

Cette phase n'a pas de fin explicite. En effet, on pourrait prendre comme signal de fin, la fin de détection d'escalier. Cependant, on constate de manière empirique que le drone cesse de voir des marches dans son champ de vision au moins une dizaine de marches avant la fin de l'escalier. On pourrait également mettre en place un timer pour finir de monter l'escalier mais l'angle des escaliers étant variant, on ne saurait quand s'arrêter. Il est donc à la charge de l'utilisateur de changer de mode lorsque cela lui semble pertinent. En l'absence d'autre commande, le drone continuera d'avancer dans la même direction.

4.4.3.4 Problèmes rencontrés et solutions

Lissage de la position du point visé au cours du temps

Le premier problème était que le milieu de l'escalier où du trapèze était instable du fait de la détection de segments différents. Nous avons donc lissé la position de ce point. Nous avons effectué des mesures en simulation et dans la réalité.

En simulation, il y a peu de bruit et les lignes détectées varient peu. Nous pouvons cependant remarquer que le centre de l'escalier et le centre du trapèze par régression sont plus stables que celui de la régression par svm.

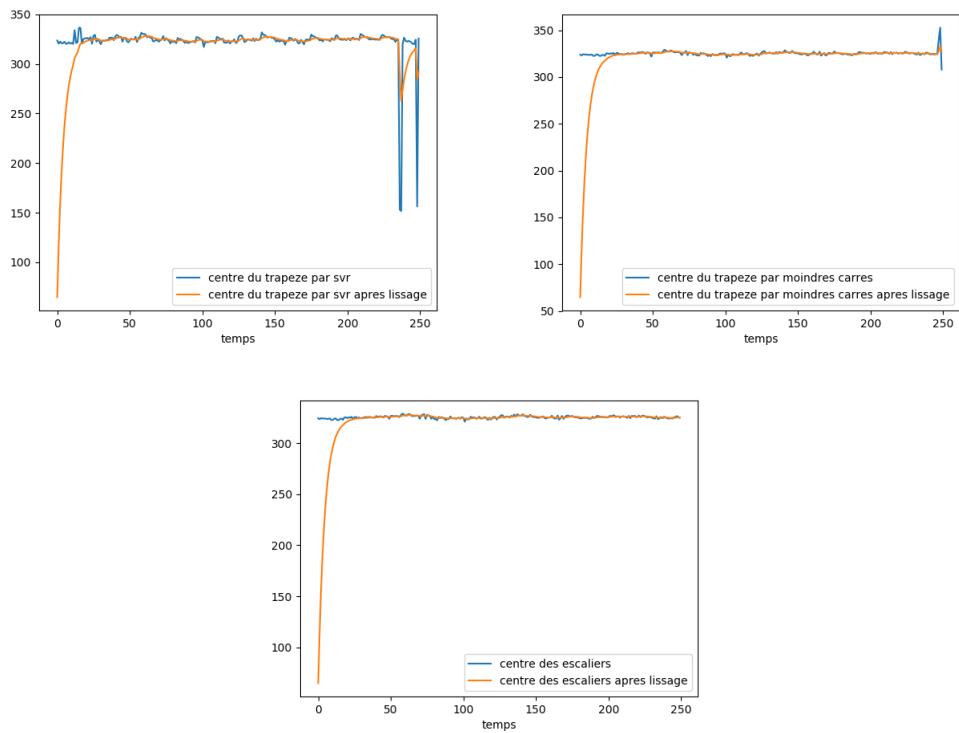


Figure 66 – Lissage du target point pour régression SVR en haut à gauche, Trapèze en haut à droite, régression linéaire en bas

Cependant dans la réalité, cela change complètement. On peut remarquer sur les figures suivantes que les courbes de l'abscisse du centre du trapèze calculé avec une SVM sont similaires à celles de l'abscisse du centre de l'escalier. De plus, le centre du trapèze calculé par régression linéaire simple diverge. Cela peut ne pas être flagrant (figure 68) mais sur d'autres exemple cela se voit clairement (figure 68). Sur les figures suivantes, on peut voir ces 3 courbes pour deux enregistrements différents. En haut à gauche la figure correspond au tracé de l'abscisse du centre du trapèze calculé par svr, au milieu du trapèze par régression linéaire et à droite le centre de l'escalier.

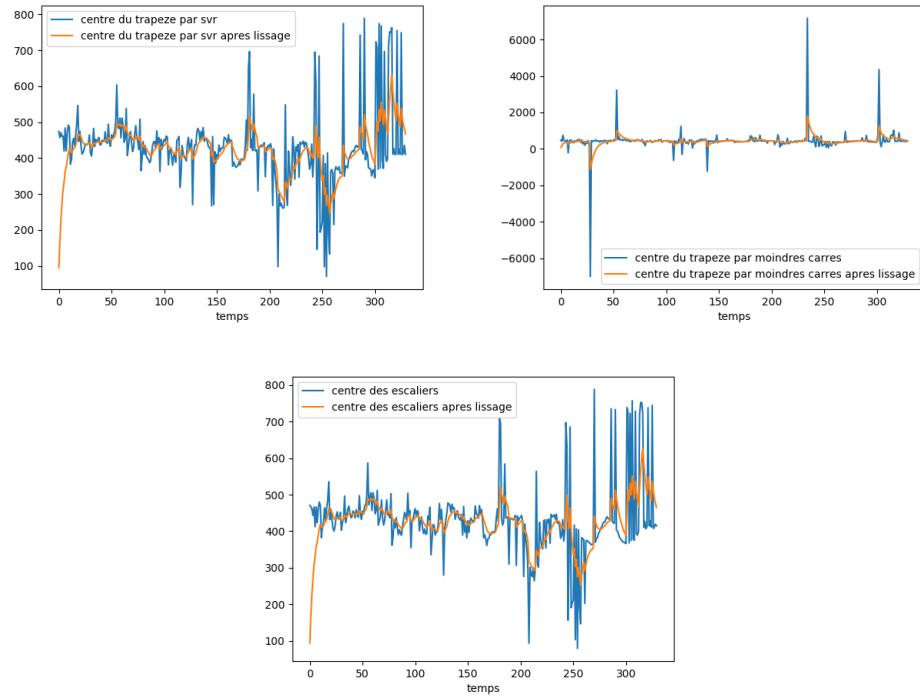


Figure 67 – Exemples de relevés de l'abscisse du milieu de l'escalier au cours du temps, avant et après lissage. Enregistrement 1

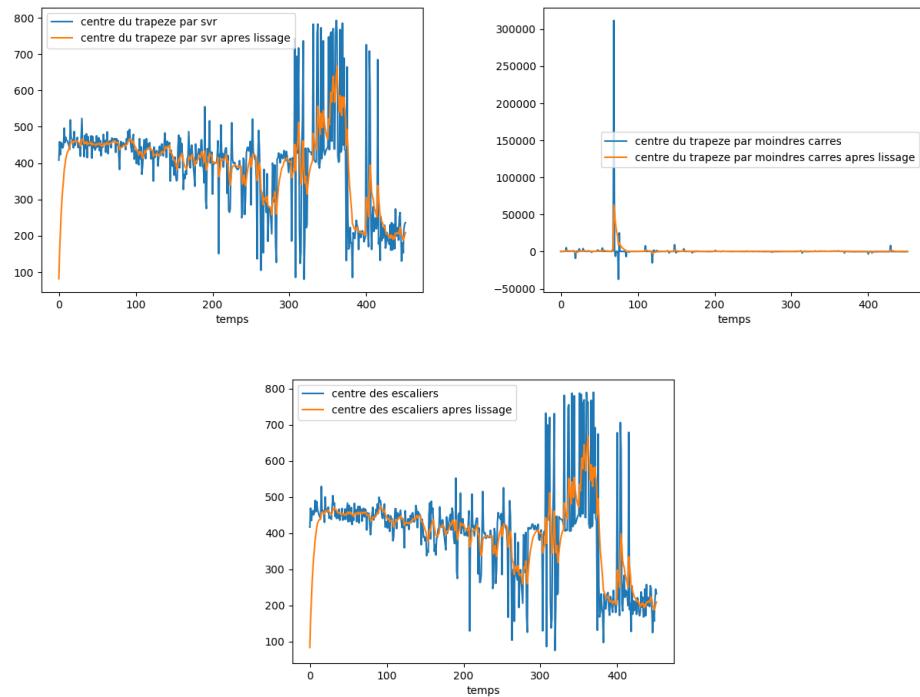


Figure 68 – Exemples de relevés de l'abscisse du milieu de l'escalier au cours du temps, avant et après lissage. Enregistrement 2

Lissage de l'odométrie Une fois ces lissages réalisés, nous avons tout de même constaté des écarts soudains dans le comportement du drone en phase de monté, nous avons ensuite corrélé ces écarts avec des variations brusques du signal en sortie de l'odométrie (qui intervient dans les computeCommand). Pour pallier cela, nous avons entrepris de lisser l'odométrie (en effet la pente maximale en vitesse est limitée par la consigne maximale en vitesse).

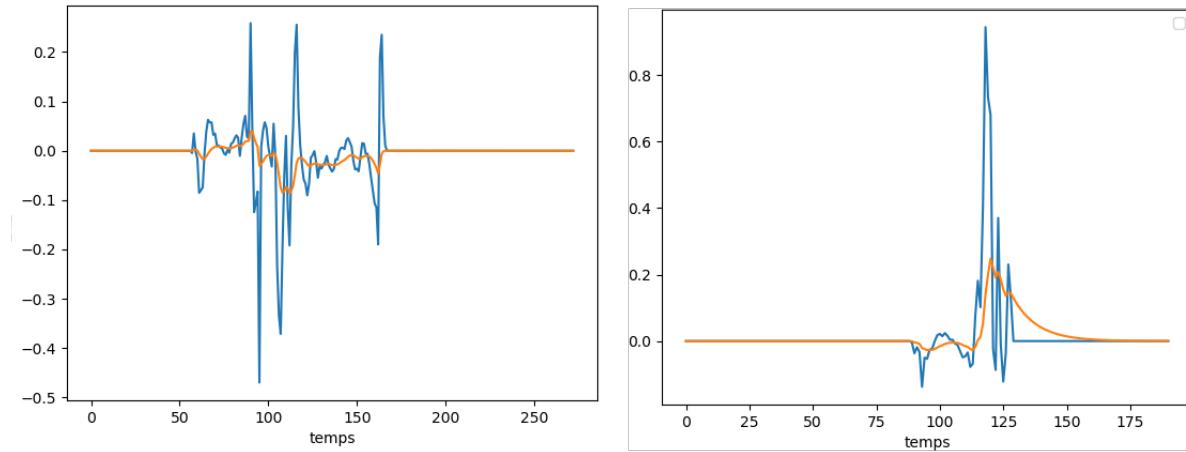


Figure 69 – Lissage de l'odométrie

Nous n'avons pas pu mettre en application ce dernier lissage et déterminer si cela suffisait à stabiliser le drone car les drones n'étaient plus fonctionnels lorsque l'on a détecté ce problème.

4.5 Commande pour la recherche des portes ouvertes

4.5.1 Introduction

Le but de ce projet est d'assister à la navigation de drone en intérieur, d'où la nécessité d'ajouter une commande pour détecter et traverser les portes ouvertes. Le principe de la commande est expliqué dans la figure suivante :

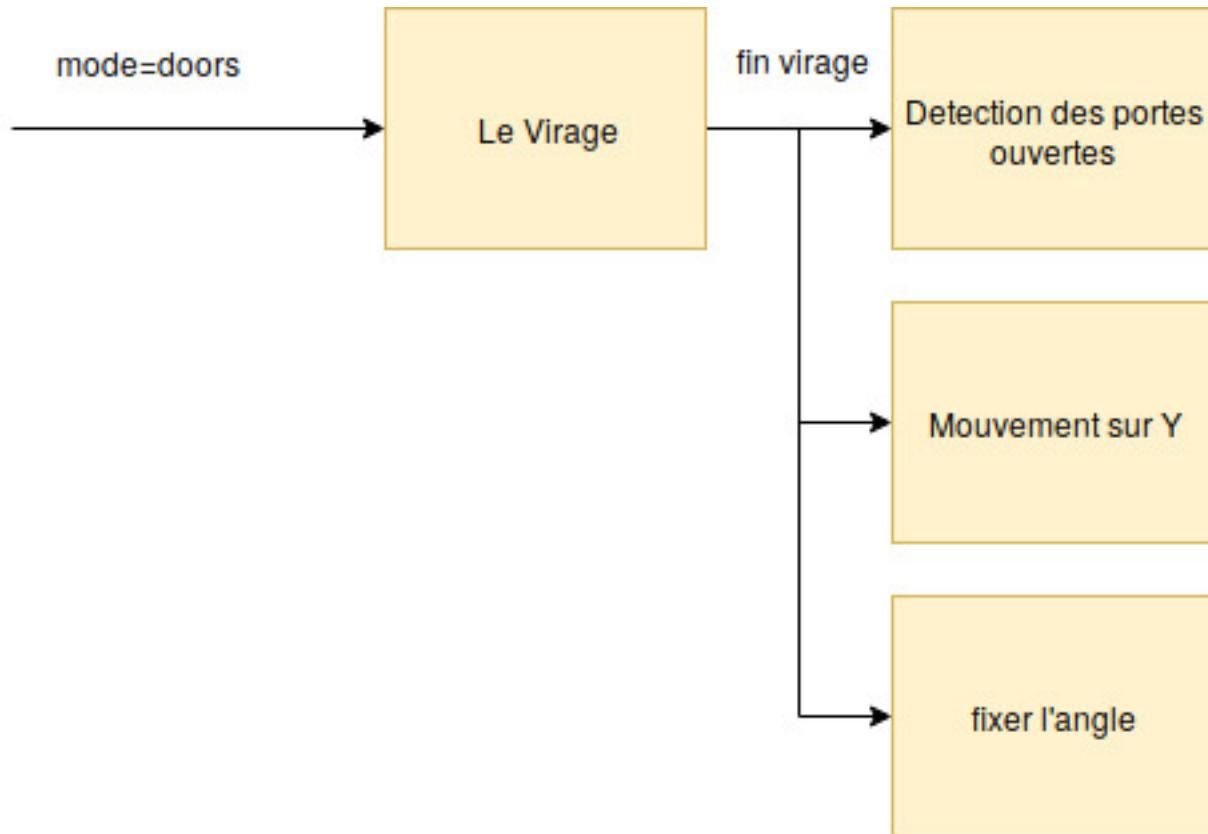


Figure 70 – Commande pour les portes

4.5.1.1 Étape 1

Après avoir reçu la commande de recherche de portes, le drone avance pendant un temps constant, puis commence à tourner à gauche (à droite), par exemple, tout en avançant. Lorsque ce tour est terminé, le drone se déplace à une vitesse constante le long de l'axe y positif (négatif).

4.5.1.2 Étape 2

Après avoir effectué le virage, le drone doit garder l'angle de rotation en se déplaçant sur Y. C'est le rôle du module expliqué dans le paragraphe précédent. Le but de fixer l'angle de rotation de drone lors du mouvement sur Y est de garder une direction de mouvement parallèle au mur. Le mouvement sur Y permet d'estimer la distance en se basant sur le flux optique.

5 Le Séquenceur

5.1 Introduction

Pour que le projet fonctionne comme un seul élément, nous devons assembler toutes les briques de construction et les coordonner à l'aide d'un contrôleur logique. C'est le rôle du séquenceur. L'idée est inspirée du célèbre grafctc utilisé pour programmer les PLCs.

5.2 Structure du projet

Le projet contient un ensemble de nœuds, qui possèdent chacun un nom qui leur est unique. Ces noeuds s'activent lorsqu'ils reçoivent un signal d'activation qui contient leur nom.

Certains noeuds possèdent une autre propriété qui leur permet d'envoyer des signaux de fin de tâche.

Ce signal prend la forme du nom du noeuds, envoyé sur un topic précis.

5.3 Le Grafcet

D'après Wikipedia :

Le Grafcet (Graphe Fonctionnel de Commande des Étapes et Transitions) est un mode de représentation et d'analyse d'un automatisme, particulièrement bien adapté aux systèmes à évolution séquentielle, c'est-à-dire décomposable en étapes. Il est dérivé du modèle mathématique des réseaux de Petri.

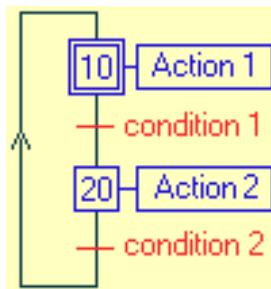


Figure 71 – Exemple de Grafcet

Le mode de représentation normalisé par l'Union technique de l'électricité (UTE) est le suivant :

1. Une étape est représentée par un carré repéré par un numéro identificateur. Une étape active peut être désignée par un point au-dessous du numéro. Les actions associées sont indiquées de façon littérale ou symbolique, dans un rectangle relié à la partie droite. Une étape initiale est représentée par un carré doublé. (En bleu, fig. au-dessus)

2. Une liaison orientée est représentée par une ligne, parcourue par défaut de haut en bas ou de gauche à droite. Dans le cas contraire, on utilise des flèches. On évite les croisements. (En noir, fig. au-dessus)
3. Une transition entre deux étapes est représentée par une barre perpendiculaire aux liaisons orientées qui la relient aux étapes précédente(s) et suivante(s). Une transition indique la possibilité d'évolution entre étapes. À chaque transition est associée une réceptivité inscrite à droite de la barre de transition. Une réceptivité est une condition logique qui permet de distinguer parmi toutes les combinaisons d'informations disponibles celle qui est susceptible de faire passer le système aux étapes suivantes (En rouge, fig. au-dessus).

5.4 Convention de nomenclature

5.4.1 Le séquenceur

Le séquenceur est le contrôleur logique qui active les noeuds par un ordre préétabli par l'utilisateur. Ce séquencement logique est pareil à celui du grafctet.

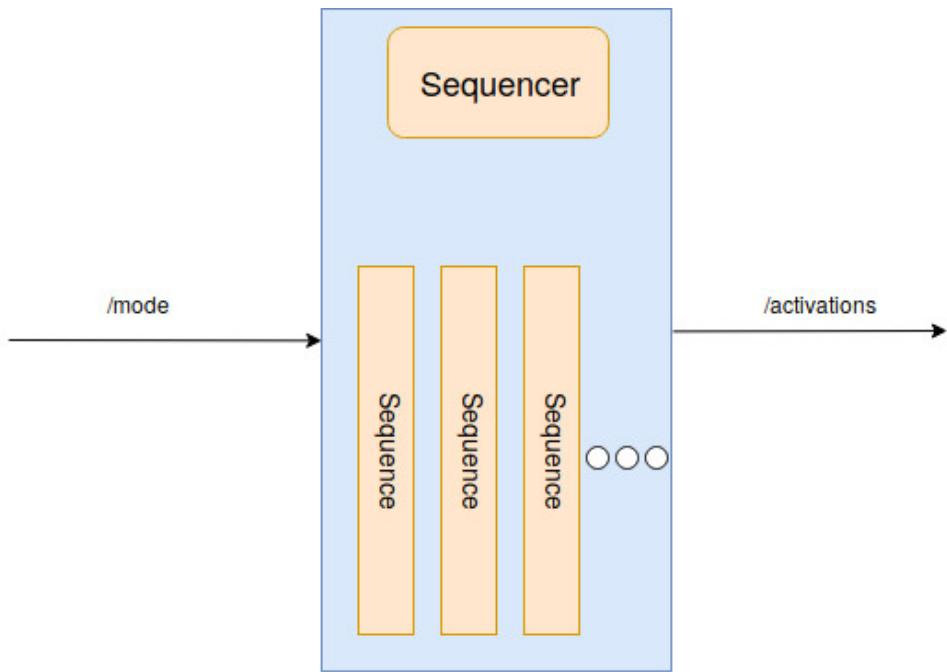


Figure 72 – Le séquenceur

Un séquenceur prend comme entrée le mode envoyé par la télécommande et sort les signaux d'activations aux noeuds. Dans le séquenceur on trouve des séquences prédéfinies et on peut également ajouter des séquences (voir README du git).

5.4.2 La séquence

Par définition une séquence est une entité divisée en 2 parties : le nom de la séquence et la liste des phases qui la constituent.

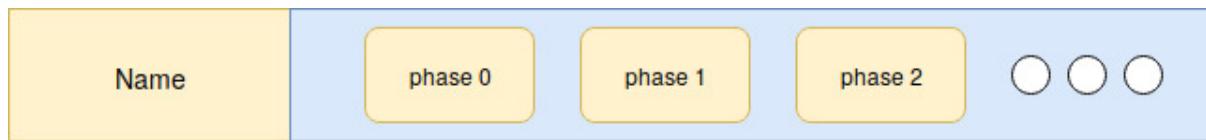


Figure 73 – La Séquence

5.4.3 La phase

Une phase est un ensemble de noeuds à activer et des conditions à attendre pour sortir de cette phase et passer à la phase suivante.// Les conditions ne sont que les signaux de fin de tâche envoyés par les noeuds activés.

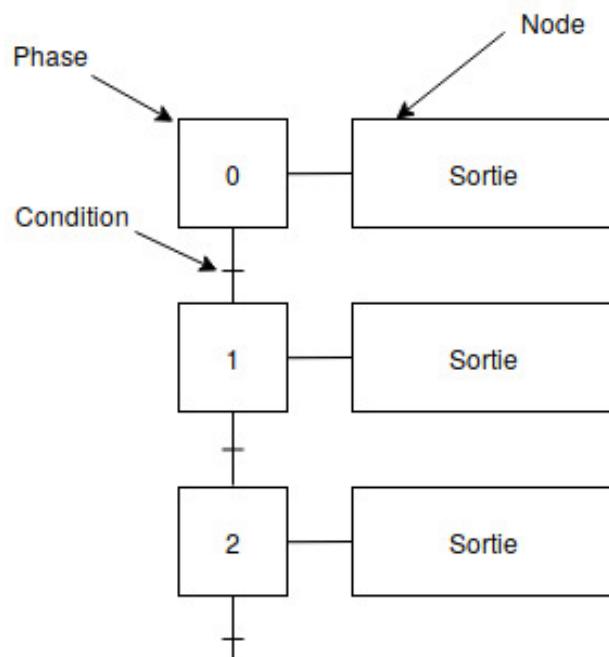


Figure 74 – Une Phase

5.4.4 Le projet en grafset

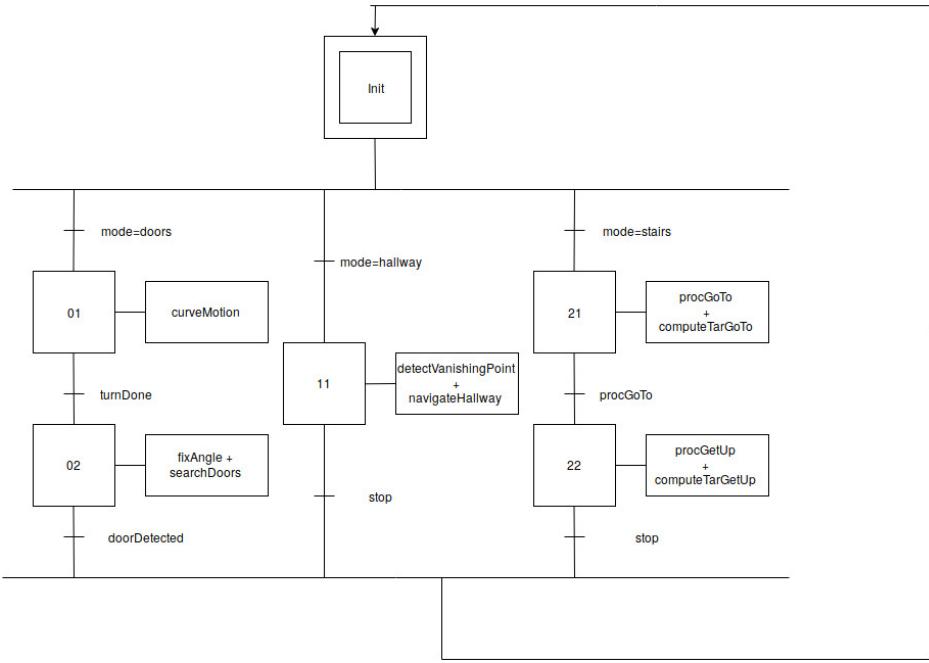


Figure 75 – Le projet en grafset



Figure 76 – Une Séquence

Lorsqu'on lance le projet, le séquenceur est dans la partie Init, où il attend un mode pour savoir quelle séquence lancer.

Supposons que la télécommande envoie un mode "doors", le séquenceur lance alors la séquence qui possède le nom "doors" (à gauche dans la figure 75), qui est présentée dans la figure 78, il active ensuite le nœud qui possède le nom "curveMotion", puis attend l'arrivée du signal l'informant que le virage est terminé. Une fois reçu, il passe à la phase 2 qui consiste à activer les nœuds responsables de la recherche de porte et à fixer le drone à un angle donné et le bouger avec la vitesse correspondante.

Une fois qu'une porte ouverte est trouvée, le séquenceur quitte la phase 2 et repasse à la phase initiale en attendant la prochaine commande.

5.4.5 La télécommande

Pour pouvoir envoyer le mode désiré au drone, nous avons programmé un simple nœud graphique qui joue le rôle d'une télécommande.

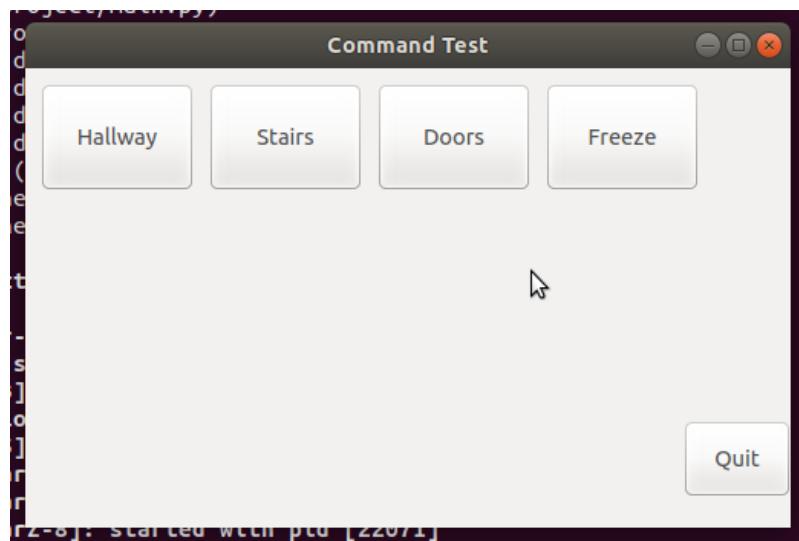


Figure 77 – La télécommande

5.4.6 Le graphe du projet

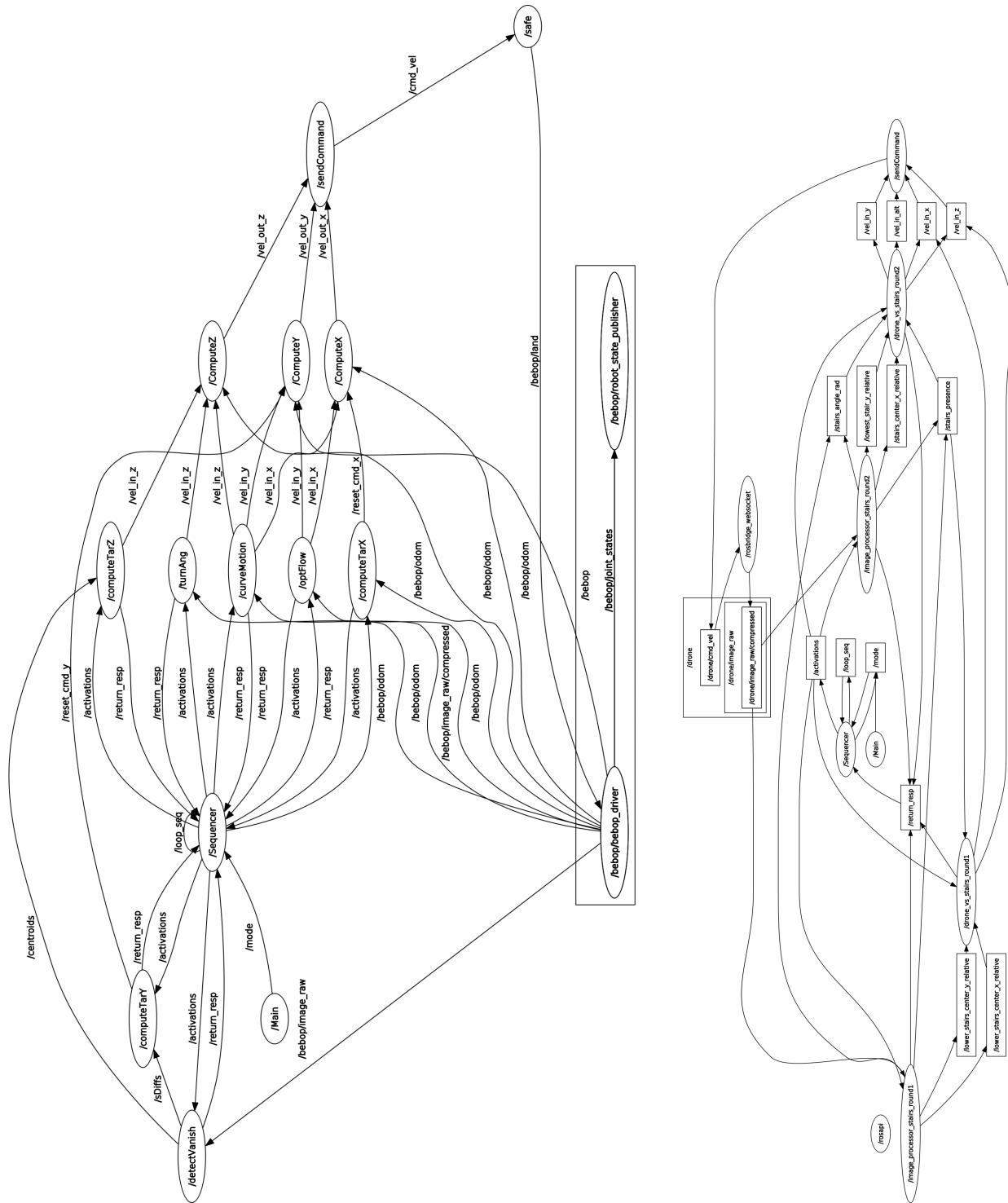


Figure 78 – Le ROS Graph

6 Appendice

6.1 Appendice A

6.1.1 Simulations

6.1.1.1 Introduction

Dans cette section, nous allons expliquer comment lancer la simulation sous l'éditeur Unity3D. Avant de commencer, notez que pour ce faire, Unity3D utilise le websocket de Rosbridge pour envoyer et recevoir des données de ROS.

6.1.1.2 Lancement ROSCORE

Vous devez d'abord bien sûr démarrer le roscore après avoir faire le sourcing suivant.

```
1 $ . <PATH_OF_CATKIN_WS>/devel/setup.bash
```

Et maintenant, tapez la commande roscore :

```
1 $ roscore
```

6.1.1.3 Lancement ROSBRIDGE

Et maintenant, vous devez lancer le serveur Rosbridge pour connecter Unity3D et le ROS à l'aide du socket Web. Pour ce faire, vous devez d'abord ouvrir un nouveau terminal, identifier la configuration de développement et exécuter la commande suivante :

```
1 $ roslaunch rosbridge_server rosbridge_websocket.launch
```

6.1.1.4 Éditeur Unity3D

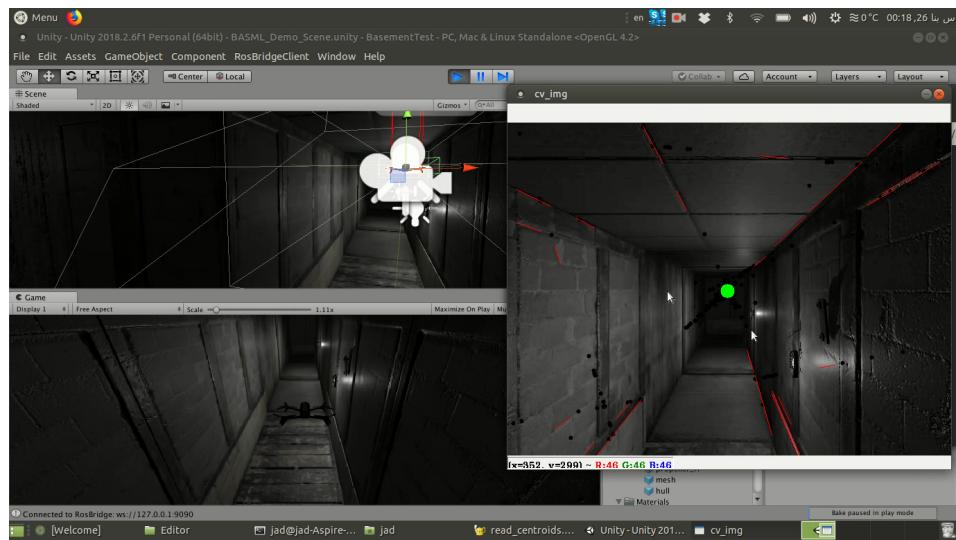
Vous devez maintenant lancer votre éditeur Unity3D et ouvrir votre projet créé après avoir suivi la procédure. ST5: Intégration des méthodes d'optimisation de la navigation autonome de drones
Et maintenant, passez au mode de jeu. Vous remarquerez que vous êtes connecté au bas de l'écran avec le ROSBridge, comme dans le terminal que vous avez utilisé pour lancer le ROSBridge.

6.1.1.5 ROSLaunch pour votre projet

Il s'agit de la dernière étape, qui consiste à exécuter votre fichier de lancement (fichier .launch) à l'aide de la commande suivante :

```
1 $ roslaunch [your package] [your.launch file]
```

6.1.1.6 Résultats



6.2 Appendice B

6.2.1 Connection au vrai drone

6.2.1.1 Introduction

Dans cette section, nous parlerons de la connexion au drone pour l'implémentation du code, celui utilisé dans ce projet est le Parrot® Bebop.



Figure 79 – Drone

6.2.1.2 Getting Ready

Afin de contrôler un vrai drone, vous devez d'abord suivre ces étapes.

6.2.1.3 Étapes

1. Compilez bebop_autonomy à partir des sources en suivant les étapes de cette procédure.
Lien
2. Clonez le package de demo_teleop créé par M.Frezza-Buet

```
1 $ git clone https://github.com/HerveFrezza-Buet/demo-teleop.git
```

3. Build votre workspace

```
1 $ cd ~/catkin_ws  
2 $ catkin build
```

4. créez dans votre /catkin/src un fichier de lancement, appelez-le par exemple dronebebop.launch, vous pouvez trouver ce fichier Ici
5. Dans votre réseau WiFi, connectez-vous au réseau de drone.

- Après avoir mis en cache le devel, comme expliqué dans l'appendice précédent, exécutez ce fichier launch.

```
1 $ cd ~/catkin_ws/src
2 $ roslaunch dronebebop.launch
```

Si vous obtenez le problème suivant :

```
[ INFO] [1550044997.548724463]: Initializing nodelet with 4 worker threads.
[ERROR] [1550044998.196578507]: Failed to load nodelet [/bebop/bebop_driver] of type [bebop_driver/BebopDriverNodelet] even after refreshing the cache: Failed to load library /home/jad/catkin_ws/devel/lib/libbebop_driver_nodelet.so. Make sure that you are calling the PLUGINLIB_EXPORT_CLASS macro in the library code, and that names are consistent between this macro and your XML. Error string: Could not load library (Poco exception = libarcommands.so: cannot open shared object file: No such file or directory)
[ERROR] [1550044998.196653982]: The error before refreshing the cache was: Failed to load library /home/jad/catkin_ws/devel/lib/libbebop_driver_nodelet.so. Make sure that you are calling the PLUGINLIB_EXPORT_CLASS macro in the library code, and that names are consistent between this macro and your XML. Error string: Could not load library (Poco exception = libarcommands.so: cannot open shared object file: No such file or directory)
[FATAL] [1550044998.196703100]: bebop_driver nodelet failed to load.
[bebop/bebop_driver-1] process has died [pid 17401, exit code 1, cmd /home/jad/catkin_ws/devel/lib/bebop_driver/bebop_driver_node __name:=bebop_driver __log:=/home/jad/.ros/log/aace7222-2f64-11e9-a9f2-18a6f70f6f07bebop-bebop_driver-1.log].
100 file: /home/jad/.ros/log/aace7222-2f64-11e9-a9f2-18a6f70f6f07bebop-bebop_driver-1* 100
```

Figure 80

Vous devez alors taper la commande suivante :

```
1 $ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/ros/melodic/lib/parrot_arsdk
```

Et relancez la commande roslaunch.

- Et maintenant, quand tout va bien, vous pouvez remarquer que lorsque vous cliquez sur le "T" (take-off), le drone décollera et "L" (land) atterrira.

En utilisant ce package, vous pouvez contrôler le drone de manière sécurisée. Si quelque chose ne va pas, il vous suffit d'appuyer sur "Espace" et le drone sera stable dans sa position.

- Et maintenant, pour tester la caméra, ouvrez un nouveau terminal et tapez ce qui suit :

```
1 $ . ~/catkin_ws/devel/setup.bash
2 $ rqt_image_view
```

Vous obtiendrez le résultat suivant :

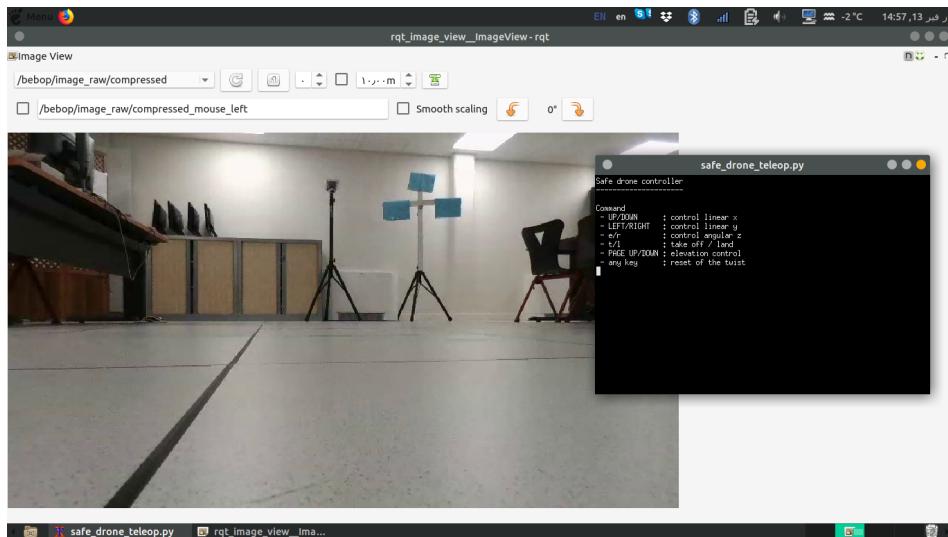


Figure 81

Si vous suivez correctement ces étapes, vous serez prêt à écrire votre propre code de contrôle et à le tester sur le vrai drone, en envoyant les commandes twist au topic /cmd_vel_in et en obtenant des images compressées à partir du topic /bebop/image_raw/compressed.

6.2.1.4 Le décollage et l'atterrissage

Après avoir exécuté la démo comme expliqué dans la section précédente, et en cliquant sur "t", le drone décollera et se stabilisera à 1 m de hauteur.



Figure 82 – Décollage

Conseil : voir Remaps !!

6.3 Appendice C

6.3.1 Filmer avec ROSBAG

Cette section explique comment enregistrer des données à partir de la caméra du drone, puis créer une vidéo à partir de ces images. Mais il faut d'abord présenter ROSBAG.

6.3.1.1 Qu'est-ce que RosBag ?

D'après le site Web de ROS :

"This is a set of tools for recording from and playing back to ROS topics. It is intended to be high performance and avoids deserialization and reserialization of the messages."

6.3.1.2 Comment utiliser RosBag

6.3.1.3 Enregistrez tous les topics :

```
1 $ rosbag record -a
```

6.3.1.4 Jouer un fichier bag spécifique :

```
1 $ rosbag play <BAG FILE>
```

Vous pouvez trouver ici et ici plusieurs exemples.

6.3.2 Enregistrement d'une Vidéo

Pour enregistrer une vidéo à l'aide de rosbag, vous devez suivre les étapes suivantes :

1. Créez deux dossiers, l'un contenant vos fichiers bagfiles et l'autre destiné aux images :

```
1 $ mkdir ~/bagfiles
2 $ cd ~/bagfiles
3 $ mkdir images
```

2. Installez ffmpeg

```
1 $ sudo apt-get install ffmpeg
```

3. Accédez à /images et exécutez la commande suivante :

```
1 $ rosrun image_view extract_images _sec_per_frame:=0.01
2 ...
3 image:=<IMAGETOPICINBAGFILE> _image_transport:=compressed
```

4. Dans un terminal séparé, exécutez la commande suivante :

```
1 $ rosbag play <BAGFILE>
```

5. Une fois la lecture terminée, vous obtiendrez le dossier /images pleins d'images que vous devez utiliser pour créer une vidéo. Pour cela, vous devez utiliser la commande ffmpeg.

6. Déterminez le FPS du fichier rosbag. Utilisez info rosbag, puis divisez le nombre de messages pour le sujet de l'image par la durée (en secondes). Cette étape est très importante.
7. Lancez la commande suivante :

```
1 ffmpeg -r <FPS> -i frame%04d.jpg -b 9600 <OUTPUT>.avi
```