Projet Target Drone

# Compte rendu de projet

Groupe D2

Table des matières

[Introduction 3](#_Toc511984603)

[Répartition du travail au sein de l’équipe 4](#_Toc511984604)

[Environnement de travail 5](#_Toc511984605)

[Méthode de suivi par couleur 6](#_Toc511984606)

[Détection de la couleur 6](#_Toc511984607)

[Comportement du drone 10](#_Toc511984608)

[Méthode de détection de visage HAAR Cascade 15](#_Toc511984609)

[Méthode de détection de forme HOG 16](#_Toc511984610)

[Méthode de détection par Deep Learning 17](#_Toc511984611)

[Conclusion 19](#_Toc511984612)

# Introduction

Dans le cadre du projet S8 de notre deuxième année d’enseignement, notre équipe a choisi de travailler sur le projet de suivi de personne par asservissement visuel d’un drone Bebop II de l’entreprise Parrot. Différentes méthodes réalisées durant le projet et cherchant à respecter les critères imposés par le cahier des charges seront présentées dans ce rapport. Ces méthodes sont : détection de couleur, reconnaissance de formes et par Deep Learning. Le délai de réalisation du projet est de cinq semaines, allant du 26 mars au 27 avril. Le drone présente l’avantage particulier de fonctionner avec le Middleware Robotic Operating System (ROS), ce qui offre une très grande versatilité quant à l’implémentation de différents programmes permettant son pilotage.

Ceci impliqua ainsi la nécessité de découvrir le fonctionnement de ce middleware et de savoir comment user de son potentiel pour l’implémentation de nos méthodes de détection et de la réaction comportementale du robot selon les instructions requises. Outre le système ROS, il était indispensable d’utiliser la librairie OpenCV pour le traitement d’image. Cette bibliothèque graphique contient de nombreuses fonctions qui nous ont permis de réaliser la grande majorité des méthodes nécessaires à l’aboutissement du projet. Il était également nécessaire d’avoir des connaissances préalables sur les langages de programmation C++ ou Python, tous deux étant parfaitement adaptés pour fonctionner sur l’environnement ROS.

# Répartition du travail au sein de l’équipe

Le tableau suivant réfère la répartition des différentes tâches menant à l’aboutissement du projet effectuées par les membres de l’équipe.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Membres de l’équipe | Jingwei ZHANG | Sisong XU | Nicolas PAPET | Rodolphe LATOUR |
| Préparation de l’environnement de travail (installation de Linux Ubuntu 16.04 LTS, ROS Kinetic Kame…) | X | X | X | X |
| Création du nœud gérant le comportement du drone pour le suivi par couleur | X | X |  |  |
| Création du nœud gérant la détection de couleur |  |  |  | X |
| Implémentation du choix de la couleur détectée par clic | X |  |  |  |
| Implémentation du choix de la zone de détecter par clic |  | X |  |  |
| Création du nœud gérant la détection de forme – HAAR Cascade | X |  |  |  |
| Création du nœud gérant la détection de forme – HOG DPM |  | X |  |  |
| Création du nœud gérant le comportement du drone pour la détection de couleur | X | X |  |  |
| Création du nœud gérant le comportement du drone pour la détection de forme |  |  | X |  |
| Conception de l’architecture permettant l’utilisation du Deep Learning |  |  |  | X |
| Ajout des fonctionnalités permettant l’implémentation du Deep Learning dans le nœud de commande du drone |  |  |  | X |

# Environnement de travail

## **ROS KINETIC**

Le principe de base de ROS est de faire fonctionner en parallèle un grand nombre d’exécutables qui doivent pouvoir échanger de l’information de manière synchrone ou asynchrone. Dans notre projet, ROS permettra par exemple d’interroger à une fréquence définies les capteurs du robot, de récupérer les informations, de les traiter, de les passer à des algorithmes de traitement et de contrôler les moteurs en retour.

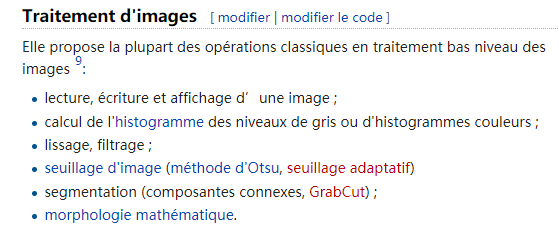
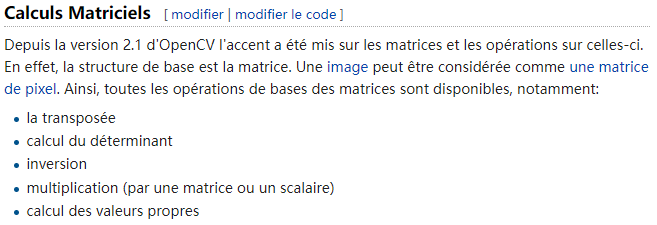
L’ensemble des traitements concernant le contrôle/commande du Drone en asservissement visuel sera effectué sous une plateforme ROS (Robot Operating System). Le système ROS comporte des outils et de librairies de robotiques mobiles qui visent à simplifier le contrôle/commande du Drone.

## **Linux Ubuntu 16.04 LTS**

|  |  |  |
| --- | --- | --- |
|  | ROS Version | Ubutnu Version |
| 2016.3 | ROS Kinetic Kame | Ubuntu 16.04 (Xenial) / Ubuntu 15.10 (Wily) |
| 2015.3 | ROS Jade Turtle | Ubuntu 15.04 (Wily) / Ubuntu LTS 14.04 (Trusty) |
| 2014.7 | ROS Indigo Igloo | Ubuntu 14.04 (Trusty) |
| 2013.9 | ROS Hydro Medusa | Ubuntu 12.04 LTS (Precise) |
| 2012.12 | ROS Groovy Galapagos | Ubuntu 12.04 (Precise) |

ROS Kinetic Kame est principalement destiné à la version Ubuntu 16.04 (Xenial), donc nous travaillons sous Ubuntu 16.04 LTS.

## **OPENcv 3.2**

L’OpenCV est une bibliothèque graphique libre qui contient des bibliothèques graphiques visant principalement la vision temps réel. Elle contient de nombreuses fonctions qui nous ont permis de réaliser la grande majorité des méthodes nécess aires à l’aboutissement du projet. Par exemple :

# Méthode de suivi par couleur

Cette étape fondamentale à la compréhension du fonctionnement de l’intégralité des outils à notre disposition fut certainement la méthode la plus simple à implémenter. Deux nœuds, détection de la couleur et comportement du drone, furent utilisés pour réaliser cette méthode.

## Détection de la couleur

Afin de bien saisir le fonctionnement du nœud, chaque étape du code sera expliquée et détaillée.

Dans un premier temps, il faut implémenter toutes les librairies contenant les fonctions que nous utilisons. Ces fonctions permettent de transmettre les différents types de messages à travers des topics dans le système ROS ainsi qu’exploiter l’image reçue du drone.

#include <ros/ros.h>

#include <image\_transport/image\_transport.h>

#include <cv\_bridge/cv\_bridge.h>

#include <sensor\_msgs/image\_encodings.h>

#include <opencv2/imgproc/imgproc.hpp>

#include <opencv2/highgui/highgui.hpp>

#include <geometry\_msgs/Point.h>

#include <fstream>

**using** **namespace** cv**;**

**using** **namespace** std**;**

On définit les valeurs de Hue, Saturation et Value nous permettant de récupérer ici une couleur orange.

int iLowH **=** 0**;**

int iHighH **=** 30**;**

int iLowS **=** 120**;**

int iHighS **=** 235**;**

int iLowV **=** 120**;**

int iHighV **=** 255**;**

Trois fenêtres sont affichées lors de l’exécution du nœud : OPENCV\_raw affiche l’image brute reçue du drone, OPENCV\_HSV affiche l’image en HSV (en français TSL : Teinte Saturation Luminosité) et OPENCV\_thresholded affiche l’image binarisée avec en blanc la couleur cible. L’utilisation d’une image HSV au lieu d’une image RVB (rouge vert bleu) permet une calibration de la couleur à détecter plus facile.

static const std**::**string OPENCV\_raw **=** "raw image"**;**

static const std**::**string OPENCV\_HSV **=** "HSV image"**;**

static const std**::**string OPENCV\_thresholded **=** "Thresholded image"**;**

Le programme est écrit dans une classe nommée ImageConverter. Un NodeHandle est défini pour initialiser le nœud et pour pouvoir faire les Subcribers, Publisher et Advertiser nécessaires. Ensuite, on définit l’objet it\_ de la classe ImageTransport qui nous permet de publier et souscrire à des messages de type image\_transport. En l’occurrence, ici, le flux vidéo que l’on veut recevoir. On définit également un Publisher publ qui servira à transmettre la position du barycentre et le nombre de pixel blanc sur l’image binarisée.

class ImageConverter

**{**

ros**::**NodeHandle nh\_**;**

image\_transport**::**ImageTransport it\_**;**

image\_transport**::**Subscriber image\_sub\_**;**

image\_transport**::**Publisher image\_pub\_**;**

ros**::**Publisher publ**;**

public**:**

ImageConverter**()**

**:** it\_**(**nh\_**)**

**{**

Ici, nous nous abonnons au topic /bebob/image\_raw qui correspond à l’image brute transmise par le Bebop II. On annonce ensuite que le nœud publiera des messages sur /image\_converter/output\_video de type image\_transport et des messages sur /cible de type geometry\_msgs.

// Subscrive to input video feed and publish output video feed

image\_sub\_ **=** it\_**.**subscribe**(**"/bebop/image\_raw"**,** 1**,**

**&**ImageConverter**::**imageCb**,** **this);**

image\_pub\_ **=** it\_**.**advertise**(**"/image\_converter/output\_video"**,** 1**);**

publ **=** nh\_**.**advertise**<**geometry\_msgs**::**Point**>(**"/cible"**,** 1000**);**

Nous définissons et nommons nos trois fenêtres qui dépendront de la bibliothèque openCV.

cv**::**namedWindow**(**OPENCV\_raw**);**

cv**::**namedWindow**(**OPENCV\_HSV**);**

cv**::**namedWindow**(**OPENCV\_thresholded**);**

**}**

Nous détruisons nos trois fenêtres si l’on appelle **~**ImageConverter**()**dans le main du programme.

**~**ImageConverter**()**

**{**

cv**::**destroyWindow**(**OPENCV\_raw**);**

cv**::**destroyWindow**(**OPENCV\_HSV**);**

cv**::**destroyWindow**(**OPENCV\_thresholded**);**

**}**

On définit la fonction imageCb qui contient l’intégralité des fonctions permettant le traitement de l’image. Elle reçoit en entrée la vidéo du drone qui est convertie pour être utilisée par la librairie d’OpenCV.

void imageCb**(**const sensor\_msgs**::**ImageConstPtr**&** msg**)**

**{**

cv\_bridge**::**CvImagePtr cv\_ptr**;**

**try**

**{**

cv\_ptr **=** cv\_bridge**::**toCvCopy**(**msg**,** sensor\_msgs**::**image\_encodings**::**BGR8**);**

**}**

**catch** **(**cv\_bridge**::**Exception**&** e**)**

**{**

ROS\_ERROR**(**"cv\_bridge exception: %s"**,** e**.**what**());**

**return;**

**}**

On récupère dans une matrice d’OpenCV imgRGB l’image cv\_ptr convertie par le module cv\_bridge afin de pouvoir effectuer les transformations nécessaires. On crée ensuite une matrice HSV qui nous permettra de détecter la couleur voulue de manière plus précise par la suite.

cv**::**Mat imgRGB **=** cv\_ptr**->**image**;**

cv**::**Mat imgHSV**;**

cv**::**cvtColor**(**imgRGB**,** imgHSV**,** cv**::**COLOR\_BGR2HSV**);**

On crée ensuite la matrice qui contiendra l’image binarisée imgThresholded.

cv**::**Mat imgThresholded**;**

Une fois ces valeurs définies, il nous reste plus qu’à préciser ce que la matrice imgThresholded devra contenir grâce à la fonction inRange.

// Perform binarisation

cv**::**inRange**(**imgHSV**,** cv**::**Scalar**(**iLowH**,** iLowS**,** iLowV**),** cv**::**Scalar**(**iHighH**,** iHighS**,** iHighV**),** imgThresholded**);**

La manipulation suivante nous permet en premier lieu de supprimer les petits objets de couleur orange pouvant survenir sur l’image et ensuite de remplir les trous de petites tailles compris dans des gros amas de pixels. Il s’agit en quelque sorte d’un effet de « blur ». Le but de ces opérations est d’avoir une image binarisée qui ne contient qu’un seul groupe de pixels adjacents, affinant la précision de notre tracking

//morphological opening (remove small objects from the foreground)

cv**::**erode**(**imgThresholded**,** imgThresholded**,** cv**::**getStructuringElement**(**cv**::**MORPH\_ELLIPSE**,** cv**::**Size**(**5**,** 5**))** **);**

cv**::**dilate**(** imgThresholded**,** imgThresholded**,** cv**::**getStructuringElement**(**cv**::**MORPH\_ELLIPSE**,** cv**::**Size**(**5**,** 5**))** **);**

//morphological closing (fill small holes in the foreground)

cv**::**dilate**(** imgThresholded**,** imgThresholded**,** cv**::**getStructuringElement**(**cv**::**MORPH\_ELLIPSE**,** cv**::**Size**(**5**,** 5**))** **);**

cv**::**erode**(**imgThresholded**,** imgThresholded**,** cv**::**getStructuringElement**(**cv**::**MORPH\_ELLIPSE**,** cv**::**Size**(**5**,** 5**))** **);**

La binarisation effectuée, il ne nous reste plus qu’à calculer la position du barycentre et le nombre de pixels blanc présent dans la matrice. On dessine également un réticule à cette position sur la matrice imgRGB pour pouvoir suivre son évolution en temps réel sur l’image RGB. Pour les deux premières étapes, la fonction Moments d’OpenCV nous permet de le faire très simplement.

//Barycentre calculation

cv**::**Moments moment **=** cv**::**moments**(**imgThresholded**,** **true);**

float surface **=** moment**.**m00**;**

float x **=** moment**.**m10**/**surface**;**

float y **=** moment**.**m01**/**surface**;**

cv**::**Point point**(**x**,** y**);**

cv**::**circle**(**imgRGB**,** point**,** 15**,** cv**::**Scalar**(**255**,** 255**,** 255**),** 2**);**

cv**::**circle**(**imgRGB**,** point**,** 1**,** cv**::**Scalar**(**0**,** 0**,** 255**),** 2**);**

Ensuite, nous affichons les trois fenêtres contenant les matrices imgRGB, imgHSV et imgThresholded et créons les fichiers stream.bmp renvoyant l’image imgRGB et position.txt renvoyant la position du barycentre et le nombre de point blanc (qui sera utilisé pour la technique de Deep Learning).

// Update GUI Window

cv**::**imshow**(**OPENCV\_raw**,** imgRGB**);**

cv**::**imwrite**(**"src/deep\_learning/src/stream.bmp"**,** imgRGB**);**

cv**::**imshow**(**OPENCV\_HSV**,** imgHSV**);**

cv**::**imshow**(**OPENCV\_thresholded**,** imgThresholded**);**

cv**::**waitKey**(**3**);**

// Output modified video stream

image\_pub\_**.**publish**(**cv\_ptr**->**toImageMsg**());**

// Print position of the barycentre

std**::**ofstream outfile **(**"src/deep\_learning/src/position.txt"**);**

outfile **<<** x **<<** "\r" **<<** y **<<** "\r" **<<** surface**;**

outfile**.**close**();**

La dernière étape consiste à émettre notre message publ dans notre topic /cible. Nous définissons nos trois valeurs à émettre sur les trois composantes x, y et z d’un message de type Point. ROS\_INFO**()** qui nous permet d’afficher dans ce nœud les valeurs qui seront émises.

//Publish position and surface

geometry\_msgs**::**Point myPos**;**

myPos**.**x **=** x**;**

myPos**.**y **=** y**;**

myPos**.**z **=** surface**;**

ROS\_INFO**(**"x : %f"**,** myPos**.**x**);**

ROS\_INFO**(**"y : %f"**,** myPos**.**y**);**

ROS\_INFO**(**"surface : %f"**,** myPos**.**z **);**

publ**.**publish**(**myPos**);**

**}**

**};**

Nous définissons ici une fonction permettant de sélectionner la couleur d’un pixel lorsque l’on clique dessus, définissant la couleur à suivre.

void getObjectColor**(**int event**,** int x**,** int y**,** int flags**,** void **\***param **=** **NULL)** **{**

CvScalar pixel**;**

IplImage **\***hsv**;**

**if(**event **==** CV\_EVENT\_LBUTTONUP**)** **{**

IplImage**\*** image**;**

image **=** cvCreateImage**(**cvSize**(**imgRGB**.**cols**,**imgRGB**.**rows**),**8**,**3**);**

// Get the hsv image

hsv **=** cvCloneImage**(**image**);**

cvCvtColor**(**image**,** hsv**,** CV\_BGR2HSV**);**

// Get the selected pixel

pixel **=** cvGet2D**(**hsv**,** y**,** x**);**

// Change the value of the tracked color with the color of the selected pixel

h **=** **(**int**)**pixel**.**val**[**0**];**

s **=** **(**int**)**pixel**.**val**[**1**];**

v **=** **(**int**)**pixel**.**val**[**2**];**

iLowH **=** **(**h**-**40**);**

iHighH **=** **(**h**+**40**);**

iLowS **=** **(**s**-**40**);**

iHighS **=** **(**s**+**40**);**

iLowV **=** **(**v**-**40**);**

iHighV **=** **(**v**+**40**);**

// Release the memory of the hsv image

cvReleaseImage**(&**hsv**);**

**}**

Nous définissons le main du programme. Il exécute d’abord l’initialisation de ce nœud nommé image\_converter. Ensuite il exécute l’objet ic de la classe ImageConverter précédemment écrit. ros**::**spin**()** permet d’exécuter ce programme en boucle.

int main**(**int argc**,** char**\*\*** argv**)**

**{**

ros**::**init**(**argc**,** argv**,** "image\_converter"**);**

cvNamedWindow**(**"raw image"**,**CV\_WINDOW\_AUTOSIZE**);**

cvSetMouseCallback**(**"raw image"**,**getObjectColor**);**

ImageConverter ic**;**

ros**::**spin**();**

**return** 0**;**

**}**

## Comportement du drone

Pour ce nœud, il sera nécessaire d’utiliser les utilitaires permettant les déplacements du drone.

#include <drone\_test/keyboard\_controller.h>

#include <iostream>

#include <ncurses.h>

#include <unistd.h>

On définit ensuite les variables qui seront utilisées dans ce programme. C’est également ici que les valeurs limites permettant le respect des conditions de déplacement sont imposées.

//position couleur

int couleur\_x**;**

int couleur\_y**;**

//distance entre couleur et drone

int distance\_max**=**100**;**

int distance\_x**=**400**;**

int distance\_y**=**200**;**

//taille image

int taille\_image**=**856**;**

//position centrale de l'ecran

//int x\_min = taille\_image\*(5/11);

//int x\_max = taille\_image\*(6/11);

int x\_min **=** 375**;**

int x\_max **=** 425**;**

//int pos y

int y\_min **=** 150**;**

int y\_max **=** 250**;**

//tracking actif a 1

int track**=**0**;**

//vitesse de deplacement

float vitesse\_x **=** 0.0**;**

float vitesse\_y **=** 0.0**;**

float taille\_h**;**

float yaw**;**

La fonction posCallback permet de récupérer les valeurs émises par le node image\_converter.

//recuperation de la position couleur

void posCallback**(**const geometry\_msgs**::**Point myPos**)**

**{**

couleur\_x**=**myPos**.**x**;**

couleur\_y **=** myPos**.**y**;**

taille\_h **=** myPos**.**z**;**

**}**

La fonction keypressed permet de vérifier si une saisie clavier a été effectuée.

int keypressed**(**void**)**

**{**

int ch **=** getch**();**

**if** **(**ch **!=** ERR**)** **{**

ungetch**(**ch**);**

**return** 1**;**

**}** **else** **{**

**return** 0**;**

**}**

**}**

Définition de la fonction main dans lequel se trouve l’intégralité du programme. On initialise les fonctions de ROS nécessaire et s’abonne au topic /cible pour récupérer les valeurs de positions et de surface.

int main**(**int argc**,** char **\*\***argv**)**

**{**

ros**::**init**(**argc**,** argv**,** "autocontroller"**);**

ros**::**NodeHandle n**;**

ros**::**Rate loop\_rate**(**30**);**

ros**::**Subscriber sub **=** n**.**subscribe**(**"/cible"**,** 1000**,** posCallback**);**

DroneController bebop**;**

float avancement **=** 0.00**,** translation **=** 0.00**,** hauteur **=** 0.00**,** rotation **=** 0.00**;**

initscr**();**

cbreak**();**

noecho**();**

nodelay**(**stdscr**,** TRUE**);**

La condition « tant que » est respectée si la connexion du nœud à ROS est établie (i.e le programme n’a pas été arrêté volontairement ou involontairement). Ensuite, nous avons dans le cas où une saisie clavier est effectuée des messages qui seront transmis au drone. Cela permet de contrôle le drone depuis ce nœud. La saisie clavier « v » permet d’activer ou désactiver l’opération de tracking de cible. Toute autre saisie clavier référencée ici permet de désactiver le tracking.

**while(**ros**::**ok**())**

**{**

**if** **(**keypressed**())** // a key is pressed

**{**

**switch(**getch**())** // ASCII values

**{**

**case** 116**:** // t: decoller

bebop**.**sendTakeoff**();**

track**=**0**;**

**break;**

**case** 110**:** // n: atterrir

bebop**.**sendLand**();**

track**=**0**;**

**break;**

**case** 105**:** // i: avancer

avancement **=** 0.1**;**

**break;**

**case** 107**:** // k: reculer

avancement **=** **-**0.1**;**

**break;**

**case** 97**:** // a: rotation gauche

rotation **=** 0.1**;**

**break;**

**case** 100**:** // d:rotation doite

rotation **=** **-**0.1**;**

**break;**

**case** 106**:** // j:translation gauche

translation **=** 0.1**;**

**break;**

**case** 108**:** // l:translation droite

translation **=** **-**0.1**;**

**break;**

**case** 119**:** // w:monter en altitude

hauteur **=** 0.1**;**

**break;**

**case** 115**:** // s:descendre en altitude

hauteur **=** **-**0.5**;**

**break;**

**case** 118**:** // v: track or stop tracking

**if(**track**!=**1**){**

track**=**1**;**

**}**

**else** **if(**track**!=**0**){**

track**=**0**;**

**}**

**break;**

**default:**

**break;**

**}**

flushinp**();** // on vide le buffer de getch

**}**

Dans le cas où aucune saisie clavier n’est effectuée, nous pouvons nous retrouver dans le cas où le tracking est activé (i.e. track**=**1). On affiche donc en boucle la position du barycentre et le nombre de pixel blanc à l’écran.

**else** // a key is not pressed

**{**

**if(**track**!=**0**)**

**{**

printf**(**"couleur X %d\n\r"**,**couleur\_x**);**

printf**(**"couleur y %d\n\r" **,**couleur\_y**);**

printf**(**"taille %f\n\r"**,**taille\_h**);**

Les conditions de déplacement sont ensuite définies. Dans le cas où le curseur serait au-delà d’un seuil défini pour les positions x max et min, et y max et min, alors le drone agira pour qu’il se retrouve dans des conditions hors de ces seuils. Il exécutera ce type de comportement pour les quatre cas. (i.e. curseur trop bas, trop haut, trop à gauche et trop à droite.) Pour les cas où la cible serait trop proche ou trop éloignée, le drone se réfère au nombre de pixels blancs à l’image. S’il est trop important, il reculera, s’il se trouve en dessous d’un seuil bas, il avancera. Et si la cible est beaucoup trop loin ou qu’aucun pixel n’est présent sur l’image, alors le drone ne bougera pas.

//calcul vitesse de deplacement en hauteur

**if(**couleur\_y**<**y\_min**){**

vitesse\_y **=** 0.1**;**

printf**(**"monter \n\r"**);**

**}else** **if(**couleur\_y**>**y\_max**){**

vitesse\_y **=** **-**0.1**;**

printf**(**"descendre \n\r"**);**

**}else{**

vitesse\_y **=** 0**;**

printf**(**"altitude OK \n\r"**);**

**}**

printf**(**"changement d'altitude %f\n\r"**,**vitesse\_y**);**

hauteur **=** vitesse\_y**;**

//calcul vitesse de deplacement en rotation

**if(**couleur\_x**<**x\_min**){**

vitesse\_x **=** 0.1**\*((**x\_min**-**couleur\_x**)/**50**);**

printf**(**"tourner gauche \n\r"**);**

**}else** **if(**x\_max**<**couleur\_x**){**

vitesse\_x **=** **-**0.1**\*((**couleur\_x**-**x\_max**)/**50**);**

printf**(**"tourner droite \n\r"**);**

**}else{**

vitesse\_x **=** 0**;**

printf**(**"ne pas tourner \n\r"**);**

**}**

printf**(**"vitesse de rotation %f\n\r"**,**vitesse\_x**);**

rotation **=** vitesse\_x**;**

//calcul vitesse de deplacement avant/arrière

**if((**taille\_h**>**2500**)&&(**taille\_h**<**10000**)){**

printf**(**"rester\n\r"**);**

avancement**=**0**;**

**}**

**if(**taille\_h**<**2500**){**

printf**(**"avancer\n\r"**);**

avancement**=**0.1**;**

**}**

**if(**taille\_h**>**10000**){**

printf**(**"reculer\n\r"**);**

avancement**=-**0.1**;**

**}**

**if(**taille\_h**<**100**){**

// a key is pressed avancement=0;

printf**(**"Cilbe perdu\n\r"**);**

**}**

printf**(**"vitesse avancement %f\n\r"**,**avancement**);**

**}**

printf**(**"\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r\n\r"**);**

**}**

bebop**.**setCommand**(**avancement**,** translation**,** hauteur**,** rotation**);**

avancement **=** 0.00**,** translation **=** 0.00**,** hauteur **=** 0.00**,** rotation **=** 0.00**;**

ros**::**spinOnce**();**

loop\_rate**.**sleep**();**

**}**

endwin**();**

**}**

# Méthode de détection de visage HAAR Cascade

# Méthode de détection de forme HOG

# Méthode de détection par Deep Learning

Le Deep Learning est une méthode de programmation permettant de résoudre des problèmes dits implicites grâce à un procédé d’apprentissage profond. Dans le cas de notre projet, le drone doit pouvoir suivre une personne. Le Deep Learning nous offre la possibilité d’identifier avec de grandes précisions ce type d’entité sur une image. L’idée étant d’effectuer un tracking sur une personne particulière, la méthode de détection par couleur nous offrait déjà ce principe. Cependant, elle était incapable de distinguer une personne d’un objet environnant. L’outil Deep Learning nous permet ainsi d’optimiser notre méthode de détection par couleur en distinguant les personnes à l’écran.

Ne voulant pas consacrer du temps attribué au projet à l’écriture d’un programme de reconnaissance de personnes, un code Python a été récupéré (source : <https://www.pyimagesearch.com/2017/09/18/real-time-object-detection-with-deep-learning-and-opencv/>, auteur : Ph.D Adrian Rosebrock). Celui-ci permet la détection de nombreux éléments de la vie quotidienne telle que des bouteilles, des chaises, des personnes... Sa grande précision et son optimisation étaient le point fort qui a retenu notre attention.

Cependant, nous nous sommes confrontés à des nombreux soucis de performances lors de son implémentation sous ROS. Il y avait un délai important entre l’instant réel et l’image calculé (entre 3 et 5 secondes selon la taille des convolutions). Ce problème était dû au fait que les images étaient enregistrées dans un Buffer de dimension trop faible. En augmentant la taille de ce Buffer, nous parvenions à obtenir des résultats d’un ordre de grandeur similaire à l’utilisation de ce programme hors ROS, mais uniquement si ce nœud était lancé seul –autrement, les performances de l’ordinateur étaient insuffisantes. Or, il nous était nécessaire d’exécuter les programmes permettant de commander le drone.

Pour résoudre cela, la démarche fut la suivante : Le nœud *image\_converter* récupère l’image provenant du *drone* et calcul le barycentre d’une couleur choisi par un clic sur l’image. L’image brute est enregistrée dans le fichier *stream.bmp* et la position du barycentre dans le fichier *position.txt*. Le programme Deep Learning récupère l’image et les informations de position puis affiche un rectangle dans lequel se trouve la cible si et seulement s’il s’agit d’une personne et que le barycentre le cible. Les positions du rectangle sont enregistrées dans un fichier *rectangle.txt* et émise dans le nœud de contrôle *main\_controller* qui traite les informations et émet les ordres de déplacement au *drone*. Le schéma ci-dessous résume le fonctionnement :

Drone

image\_converter.cpp

position.txt

stream.bmp

real\_time\_object\_detection.py

rectangle.txt

main\_controller.cpp

Drone

[EXPLICITATION DES RESULTATS]

# Conclusion