

# **SLAM monoculaire python pour drone**

Architecture matérielle et logicielle pour la  
robotique  
ROB314

Yufei HU, Jianzhou MA et Yu WANG

April 11, 2022



# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>Drone TELLO</b>	<b>4</b>
<b>3</b>	<b>Les algorithmes</b>	<b>6</b>
3.1	SLAM . . . . .	6
3.2	PySLAM . . . . .	6
<b>4</b>	<b>Communication</b>	<b>9</b>
4.1	La communication de Tello via ordinateur . . . . .	9
4.2	La communication de Tello via ROS . . . . .	11
<b>5</b>	<b>Calibration et enregistrement</b>	<b>14</b>
5.1	Calibration de caméra . . . . .	14
5.2	Enregistrement de vidéo . . . . .	15
<b>6</b>	<b>Résultat et analyse</b>	<b>17</b>
6.1	Comparaison des méthodes . . . . .	17
6.2	Arrêt fréquent de l'algorithme . . . . .	22
6.3	Difficulté de rotation . . . . .	23
<b>7</b>	<b>Conclusion</b>	<b>25</b>
<b>8</b>	<b>Usage</b>	<b>26</b>
8.1	La communication de TELLO via ordinateur . . . . .	26
8.2	La communication de TELLO via ROS . . . . .	26
8.3	SLAM . . . . .	26

# 1 Introduction

L'objectif de ce projet est de tester les algorithmes SLAM existant en python en utilisant une seule caméra pour construire sa carte et se localiser à l'intérieur, et les tester pour notre drone TELLO.

Comme nous le savons, le SLAM (Simultaneous Localization and Mapping) est la technologie de pointe reconnue par l'industrie en matière de positionnement spatial dans le domaine visuel. Il est principalement utilisé pour résoudre les problèmes de positionnement et de construction de cartes des robots lorsqu'ils se déplacent dans des environnements inconnus. Les algorithmes SLAM sont souvent utilisés pour résoudre nos problèmes de robotique mobile et de navigation autonome.

Mais la plupart de ces SLAM sont très gourmands en ressources et fonctionnent avec des capteurs complexes (caméras stéréo, LIDAR 3D). Mais pour le drone TELLO, sa charge utile et sa puissance sont faibles, il ne possède donc aucune de ces caractéristiques, ce qui rend très difficile la mise en œuvre d'algorithmes SLAM sur un drone TELLO. C'est aussi la principale difficulté de ce projet.

## 2 Drone TELLO

Tello (comme l'illustre la Figure 1) est un petit quadcoptère doté d'un système de positionnement visuel et d'une caméra embarquée. Grâce à son système de positionnement visuel et à son contrôleur de vol avancé, il peut faire du surplace dans la bonne position et convient pour les vols en intérieur. Tello capture des photos de 5 mégapixels et diffuse des vidéos en direct 720p sur l'application Tello de votre appareil mobile. Il a une durée de vol maximale d'environ 13 minutes et une portée maximale de 100 m (328 pieds). La fonction de sécurité intégrée permet à Tello d'atterrir en toute sécurité même s'il perd sa connexion, et son protège-hélice peut être utilisé pour plus de sécurité.



Figure 1: Drone Tello

Les paramètres du produit sont présentés dans la Figure 2. Comme nous pouvons le voir, il ne pèse que 89 g, ne possède qu'une seule caméra et ne consomme que 10 W. Comme nous l'avons mentionné précédemment, il ne dispose pas de suffisamment de ressources et de capteurs pour exécuter l'algorithme SLAM comme un robot normal, c'est pourquoi nous avons choisi d'utiliser un algorithme SLAM monoculaire.

Aircraft (Model: TLW004)	
Weight (including Propeller Guards)	87 g
Max Speed	17.8 mph (28.8 kph)
Max Flight Time	13 minutes (0 wind at a consistent 9mph (15 kph))
Operating Temperature Range	32° to 104° F (0° to 40° C)
Operating Frequency Range	2.4 to 2.4835 GHz
Transmitter (EIRP)	20 dBm (FCC) 19 dBm (CE) 19 dBm (SRRC)
Camera	
Max Image Size	2592×1936
Video Recording Modes	HD: 1280×720 30p
Video Format	MP4
Flight Battery	
Capacity	1100 mAh
Voltage	3.8 V
Battery Type	LiPo
Energy	4.18 Wh
Net Weight	25±2 g
Charging Temperature Range	41° to 113° F (5° to 45° C)
Max Charging Power	10 W

Figure 2: Paramètres du drone Tello

## 3 Les algorithmes

### 3.1 SLAM

SLAM (Simultaneous Localization and Mapping) consiste à résoudre le problème de comment un robot se déplace dans un environnement inconnu, comment déterminer sa propre trajectoire en observant l'environnement, et construire une carte de l'environnement en même temps.

Selon les différents capteurs, il peut être divisé en:

- SLAM 2D/3D basé sur le LIDAR.
- SLAM RGBD basé sur des caméras de profondeur.
- SLAM visuel basé sur des capteurs de vision.
- SLAM basé sur un capteur visuel et une centrale inertielle.

Dans ce projet, on se concentre sur l'étude du SLAM visuel( **SLAM visuel basé sur des capteurs de vision**). En plus, on utilise l'odométrie visuelle pour estimer progressivement le mouvement de la caméra. La création d'un SLAM complet comprend l'ajout d'une détection de fermeture de boucle et d'une optimisation globale pour obtenir des cartes précises et cohérentes à l'échelle mondiale.

L'odométrie visuelle est souvent utilisée comme frontal d'un système SLAM. Sa méthode de calcul incrémentielle peut estimer le mouvement de la caméra entre les images adjacentes, mais cela signifie également que l'erreur entre les images adjacentes affectera l'estimation de trajectoire ultérieure, ce qui signifie qu'elle dérivera dans le temps. Dans le même temps, l'odométrie visuelle ne contient pas de module de cartographie. Par conséquent, on utilise un SLAM complet qui prend en compte la cohérence globale des trajectoires et des cartes des caméras, mais cela signifie également que davantage de ressources informatiques sont nécessaires pour calculer l'optimisation globale. **Par conséquent, on n'a pas implémenté le SLAM en temps réel dans ce projet.**

### 3.2 PySLAM

Dans cette partie, on présente la bibliothèque open source python de slam visuel qui est **PySLAM**<sup>1</sup>. PySLAM contient une implémentation python d'un pipeline d'odométrie visuelle (VO) monoculaire. Il supporte de nombreuses fonctionnalités locales classiques et modernes, et offre une interface pratique pour celles-ci. De plus, il rassemble d'autres outils communs et utiles de VO et SLAM.

---

<sup>1</sup><https://github.com/luigifreda/pyslam>

- **Odométrie visuelle (VO):** À chaque étape, la pose actuelle de la caméra est estimée par rapport à la pose précédente afin de récupérer une échelle inter-image correcte pour estimer une trajectoire valide.
- **SLAM:** À partir de VO, il ajoute le suivi des caractéristiques sur plusieurs images, la triangulation des points, la gestion des images clés et l'ajustement du faisceau afin d'estimer la trajectoire de la caméra à l'échelle et de construire une carte.

Dans nos expériences, nous testons d'abord le pipeline de VO sur l'ensemble de données public KITTI [Fritsch et al. \(2013\)](#). Les résultats expérimentaux sont présentés dans les figures suivantes.

La résultatat de la visualisation est présenté dans la Figure 3.



Figure 3: Visualisation du pipeline de VO sur le jeu de données KITTI

Le nombre de points correspondants et les statistiques d'erreur sont indiqués dans la Figure 4 ci-dessous.

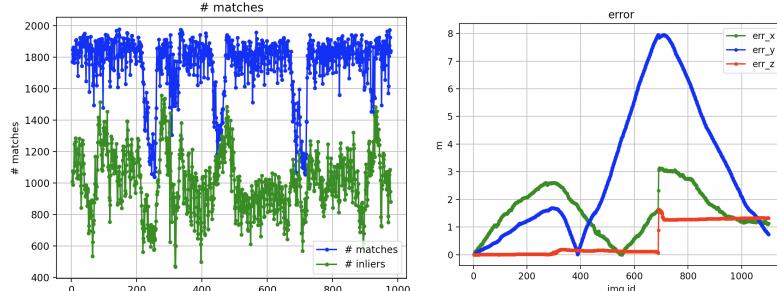


Figure 4: Statistiques sur les points de correspondance et les erreurs basées sur VO

De même, VO construit des modèles 2D et 3D pour la trajectoire de mouvement de la caméra, et les résultats sont présentés dans la Figure 4.

Comme mentionné précédemment, VO est basé sur une série d'images pour estimer la trajectoire de mouvement du robot et il n'y a pas de processus de cartographie. Par conséquent, dans les expériences suivantes, on se concentre sur le SLAM basé sur VO pour la localisation et la cartographie.

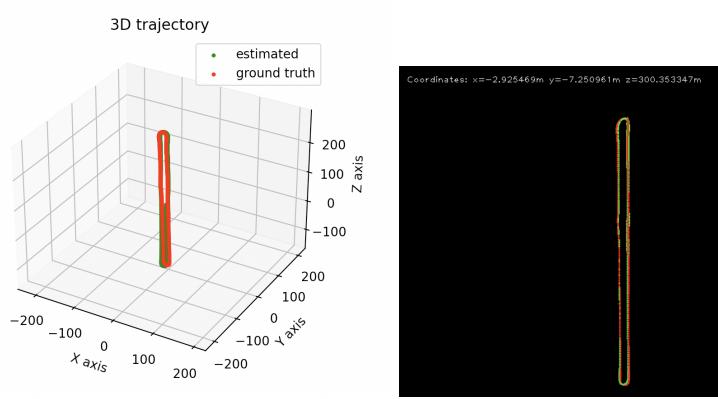


Figure 5: Tracés de trajectoire en 3-D et 2-D

## 4 Communication

### 4.1 La communication de Tello via ordinateur

Pour aider les utilisateurs à contrôler TELLO à partir d'un PC, DJI propose le SDK officiel TELLO<sup>2</sup>. Le Tello SDK se connecte à l'appareil via un port Wi-Fi UDP, ce qui permet aux utilisateurs de contrôler le drone à l'aide de commandes textuelles. Dans la documentation, il fournit la spécification de la commande correspondante et la manière dont elle est envoyée, et indique le statut de retour de chacune de ses fonctions.

Utilisant le SDK officiel de Tello, la bibliothèque DJITelloPy<sup>3</sup> fournit une interface Python un par un aux drones Tello de DJI. Cette bibliothèque présente les caractéristiques suivantes:

- Implémentation de toutes les commandes Tello
- Récupération facile d'un flux vidéo
- Recevoir et analyser les paquets d'état
- Contrôler un essaim de drones

En consultant la référence API<sup>4</sup> de la bibliothèque DJITelloPy, nous pouvons accéder à toutes les classes et méthodes disponibles. Cette bibliothèque fournit une classe appelée Tello et un certain nombre de fonctions connexes pour celle-ci (notamment la connexion, le décollage, l'atterrissement, le déplacement, l'obtention du flux vidéo, etc). Nous pouvons créer une instance de cette classe et la connecter au drone TELLO (l'ordinateur doit être connecté au Wi-Fi du drone TELLO pour réussir) pour communiquer avec le drone TELLO comme présenté dans la Figure 6.

```
from djitellopy import Tello
tello = Tello()
tello.connect()
tello.takeoff()

tello.move_left(100)
tello.rotate_counter_clockwise(90)
tello.move_forward(100)

tello.land()
```

Figure 6: Exemple code de DJITelloPy

Cette bibliothèque aussi fournit quelques exemples de code:

<sup>2</sup>[https://d1-cdn.rzerobotics.com/downloads/tello/20180910/Tello%20SDK%20Documentation%20EN\\_1.3.pdf](https://d1-cdn.rzerobotics.com/downloads/tello/20180910/Tello%20SDK%20Documentation%20EN_1.3.pdf)  
<sup>3</sup><https://github.com/damiafuentes/DJITelloPy>  
<sup>4</sup><https://djitellopy.readthedocs.io/en/latest/>

- Prendre une photo
- Enregistrer une vidéo
- Faire voler un essaim (plusieurs Tellos en même temps)
- Contrôle simple en utilisant votre clavier
- Détection du pad de mission
- Contrôle manuel complet à l'aide de pygame

Mais ce que nous voulions réaliser, c'était la possibilité de contrôler le drone à partir des images prises par le drone sur un ordinateur et de stocker le flux vidéo du drone sous forme de vidéo en même temps. Ainsi, nous avons donc ajouté la possibilité de stocker des vidéos en nous référant à son code, basé sur le code *manual-control-pygame.py* (Contrôle manuel complet à l'aide de pygame).

La raison pour laquelle nous avons choisi pygame plutôt qu'opencv est que pygame est beaucoup plus fluide à contrôler (parce que différents événements sont définis dans pygame pour les pressions de touches du clavier et les relances, et des fonctions de réponse correspondantes sont définies pour eux) (comme présenté le code ci-dessous).

```

1 def keydown(self, key):
2     """ Update velocities based on key pressed
3     Arguments:
4         key: pygame key
5     """
6     if key == pygame.K_UP: # set forward velocity
7         self.for_back_velocity = S
8     elif key == pygame.K_DOWN: # set backward velocity
9         self.for_back_velocity = -S
10    elif key == pygame.K_LEFT: # set left velocity
11        self.left_right_velocity = -S
12    elif key == pygame.K_RIGHT: # set right velocity
13        self.left_right_velocity = S
14    elif key == pygame.K_w: # set up velocity
15        self.up_down_velocity = S
16    elif key == pygame.K_s: # set down velocity
17        self.up_down_velocity = -S
18    elif key == pygame.K_a: # set yaw counter clockwise velocity
19        self.yaw_velocity = -S
20    elif key == pygame.K_d: # set yaw clockwise velocity
21        self.yaw_velocity = S
22 def keyup(self, key):
23     """ Update velocities based on key released
24     Arguments:
```

```

25         key: pygame key
26     """
27     if key == pygame.K_UP or key == pygame.K_DOWN: # set zero forward/backward velocity
28         self.for_back_velocity = 0
29     elif key == pygame.K_LEFT or key == pygame.K_RIGHT: # set zero left/right velocity
30         self.left_right_velocity = 0
31     elif key == pygame.K_w or key == pygame.K_s: # set zero up/down velocity
32         self.up_down_velocity = 0
33     elif key == pygame.K_a or key == pygame.K_d: # set zero yaw velocity
34         self.yaw_velocity = 0
35     elif key == pygame.K_t: # takeoff
36         self.tello.takeoff()
37         self.send_rc_control = True
38     elif key == pygame.K_l: # land
39         not self.tello.land()
40         self.send_rc_control = False

```

Pour exploiter le code original et mettre en œuvre la fonction d'enregistrement vidéo, nous créons un objet VideoWrite qui stockera les séquences enregistrées par le drone au format vidéo dans le fichier `./video.avi` (comme présenté le code ci-dessous).

```

1 def videoRecorder(self):
2     # create a VideoWrite object, recording to ./video.avi
3     height, width, _ = self.frame_read.frame.shape
4     video = cv2.VideoWriter('video.avi',
5                             cv2.VideoWriter_fourcc(*'XVID'),
6                             30, (width, height))

```

Après le décollage du drone TELLO, nous allons créer un objet VideoWriter pour enregistrer la vidéo. Cependant, il est important de noter que nous devons enregistrer la vidéo dans un thread séparé (comme indiqué dans le code ci-dessous). Si l'enregistrement de la vidéo et l'envoi de la commande de contrôle sont effectués dans le même thread, un conflit de ressources se produira, rendant impossible l'envoi de la commande de contrôle.

```

1 recorder = Thread(target=self.videoRecorder)
2 recorder.start()

```

## 4.2 La communication de Tello via ROS

Dans cette expérience, on essaie de communiquer avec Tello via ROS. On utilise un package qui est `tello_driver`<sup>5</sup>. Il est encapsulé sur la base d'une bibliothèque de `DJITelloPy`. À la aide de ce package, on a réalisé:

---

<sup>5</sup><https://github.com/TIERS/tello-driver-ros>

- Contrôlez le drone en temps réel par ros command.
- Obtenez le flux vidéo et acceptez et analysez les paquets d'état.
- Créez un nouveau fil pour stocker des vidéos et des images en temps réel.
- En vous abonnant au noeud tello/image\_raw/h264, on a réalisé afficher et enregistrer la vidéo capturée.

La liste des nœuds ROS du système est présenté dans la Figure 7 ci-dessous.

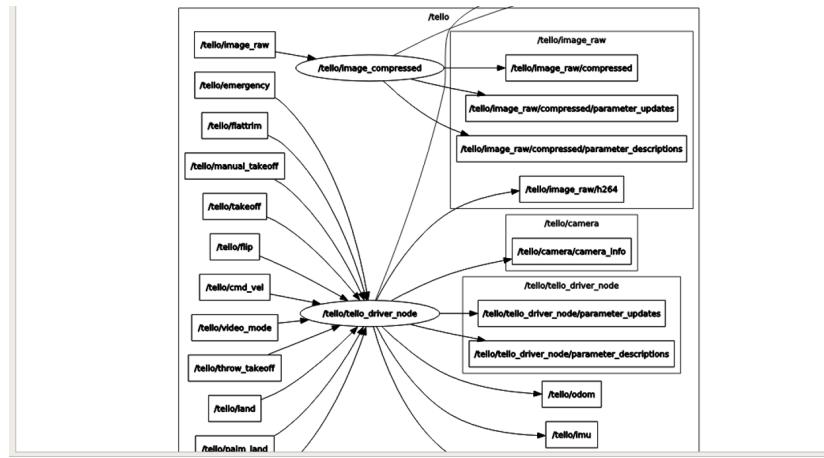


Figure 7: La liste des noeuds ROS

De plus, on utilise le terminal pour démarrer le programme et exercer le contrôle(voir la Figure 8). On utilise la commande suivante pour initier la communication avec TELLO.

```
1 roslaunch tello_driver tello_node.launch tello_ip:="192.168.10.1"
```

Pour contrôler la Tello, on doit utiliser la commande suivante:

```
1 rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/tello/cmd_vel
```

Afin de sauvegarder l'image prise par le drone, il faut s'abonner au noeud tello/image\_raw/h264 par la commande suivante:

```
1 rosrun image_view video_recorder image:=/tello/image_raw_image_transport:=h264
```

On sait que RQT est un cadre logiciel de ROS qui met en œuvre les différents outils d'interface graphique sous la forme de plugins. On a souscrit certains nœuds via RQT, comme indiqué dans la Figure 9 ci-dessous.

```

student@rosbook-03: ~
student@rosbook-03: ~ 39x29
student@rosbook-03: /home/student/drone_racing_ws/src/tello-driver-r
Tello: 09:39:17.286: Info: video data
1002648 bytes 489.5KB/sec
Tello: 09:39:19.287: Info: video data
1001264 bytes 488.6KB/sec
Tello: 09:39:21.315: Info: video data
1010996 bytes 487.1KB/sec loss=1
Tello: 09:39:23.315: Info: video data
1002875 bytes 489.7KB/sec
Tello: 09:39:25.315: Info: video data
1001407 bytes 488.9KB/sec
Tello: 09:39:27.315: Info: video data
1000190 bytes 488.4KB/sec
Tello: 09:39:29.316: Info: video data
1008705 bytes 492.4KB/sec
Tello: 09:39:31.315: Info: video data
999948 bytes 488.2KB/sec
Tello: 09:39:33.319: Info: video data
997083 bytes 486.1KB/sec
Tello: 09:39:35.320: Info: video data
1011587 bytes 493.8KB/sec
Tello: 09:39:37.320: Info: video data
998606 bytes 487.5KB/sec
Tello: 09:39:39.320: Info: video data
1005887 bytes 491.1KB/sec
Tello: 09:39:41.320: Info: video data
995334 bytes 486.0KB/sec
Tello: 09:39:43.320: Info: video data
1006272 bytes 491.2KB/sec
[ERROR] [1648712379.355246904]: Cannot send a packet to decoder
[ERROR] [1648712381.360439916]: Cannot send a packet to decoder
[ERROR] [1648712383.364578423]: Cannot send a packet to decoder

```

Figure 8: L'interface de contrôle

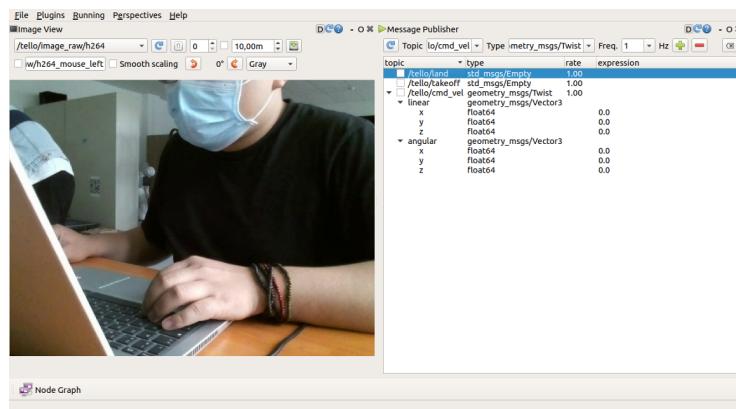


Figure 9: L'interface de RQT

## 5 Calibration et enregistrement

### 5.1 Calibration de caméra

Pour que le drone puisse estimer sa position dans le but d'établir la carte 3D, il faut d'abord calibrer sa caméra monoculaire. Pour le faire, on a imprimé un graphe de damier et pris une dizaine d'images selon des perspectives différentes avec la caméra de drone Tello. Ensuite, ces graphes sont traités par l'algorithme de calibration qui nous donnera la matrice de caméra ainsi que ces coefficients de distorsion.

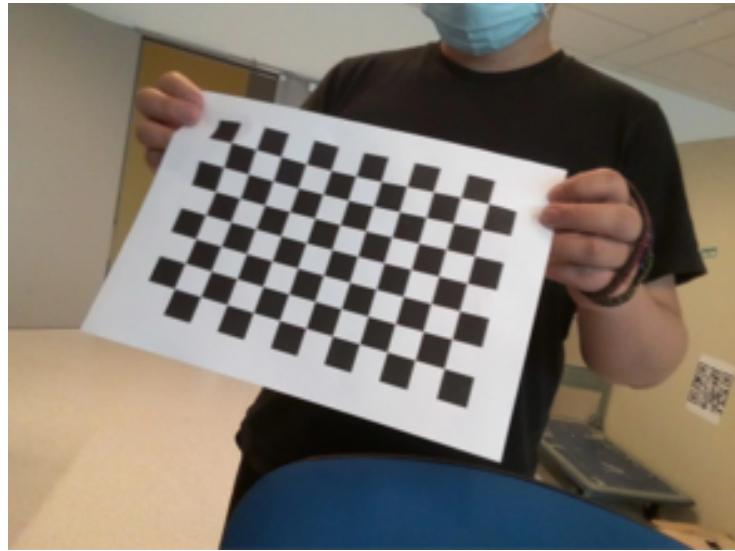


Figure 10: L'image originale de damier prise par la caméra de drone

Voici ce que l'algorithme de calibration donne. On a la matrice de caméra comme:

$$M = \begin{bmatrix} 1.95692427e + 03 & 0 & 1.29797896e + 03 \\ 0 & 1.95575859e + 03 & 1.01505392e + 03 \\ 0 & 0 & 1 \end{bmatrix}$$

Et la matrice de distorsion comme:

$$D = \begin{bmatrix} 4.50678089e - 02 \\ -2.64825754e - 01 \\ 1.49047708e - 03 \\ -3.44118748e - 03 \\ 1.54649114e + 00 \end{bmatrix}$$

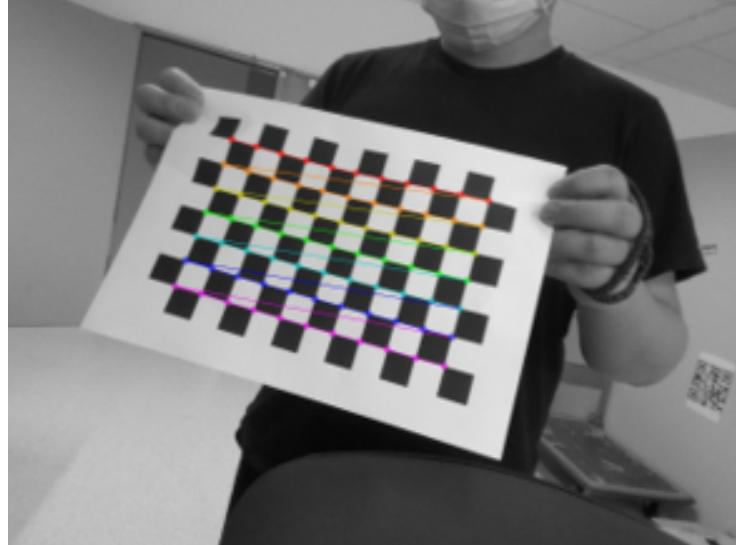


Figure 11: L'image de damier traitée par l'algorithme de calibration

La résolution de la vidéo est 1280 x 720 pixels, avec un FPS de 30, ce qui est standard.

## 5.2 Enregistrement de vidéo

Pour essayer la performance du module de PySLAM, il faut lui fournir une vidéo de test. Lorsque le drone Tello est un petit drone qui ne peut voler qu'à l'intérieur, on propose d'enregistrer des vidéos dans le bâtiment de l'ENSTA. Plusieurs vidéos sont enregistrées par le drone Tello dans des contextes différents, par exemple, le RDC de l'ENSTA, l'intérieur d'une salle et le couloir de laboratoire U2IS. On a finalement choisi le couloir comme notre contexte de SLAM car sa forme est une ligne droite, ce qui est facile à distinguer dans le nuage de points.

On a contrôlé le drone Tello par ROS pour qu'il suive une trajectoire directe dans le couloir. Le drone va aussi tourner à droite à la fin du couloir. La trajectoire attendue dans la carte 3D est donc deux lignes conjointes au coin de rotation.

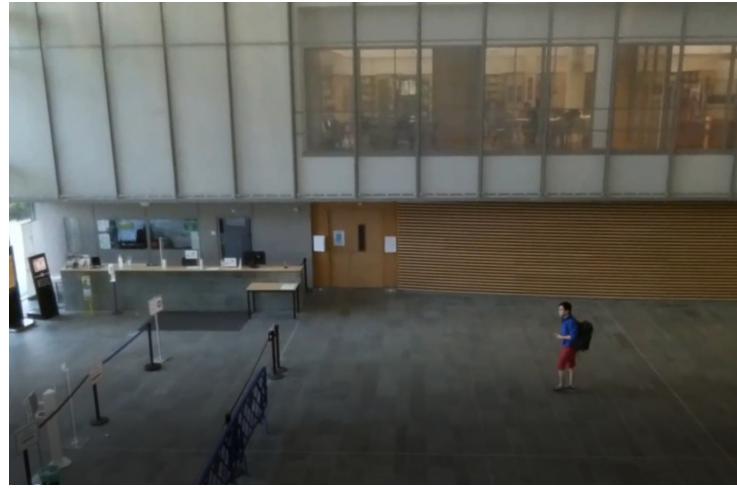


Figure 12: La vidéo enregistrée au rez-de-chaussée de l'ENSTA



Figure 13: La vidéo enregistrée au couloir d'U2IS

## 6 Résultat et analyse

### 6.1 Comparaison des méthodes

Le module PySLAM fournit plusieurs combinaisons de méthodes et leurs performances restent à tester. Ces combinaisons différentes consistent à choisir un détecteur, un descripteur et une façon d'appariement de caractéristiques afin de les intégrer dans le processus de SLAM monoculaire. Une dizaine de détecteurs et de descripteurs sont disponibles, tandis que les méthodes d'appariements sont limitées à deux: BF (Brute Force) et FLANN (Fast Library for Approximate Nearest Neighbors). Dans les essais d'algorithmes, on trouve que la méthode FLANN est plus rapide mais moins précise, puisqu'il trouve les caractéristiques les plus proches de caractéristiques au lieu du meilleur appariement. De plus, on constate que cette rapidité de FLANN ne peut pas mettre le module PySLAM en temps réel et elle cause plus d'arrêt imprévu de l'algorithme. Prenant en compte cette situation, on décide de ne tester que l'appariement BF qui calcule les distances entre toutes les caractéristiques afin de trouver un meilleur appariement.

Dix combinaisons de méthodes sont testées et évaluées avec la vidéo de test enregistrée par le drone. Voici les détails:

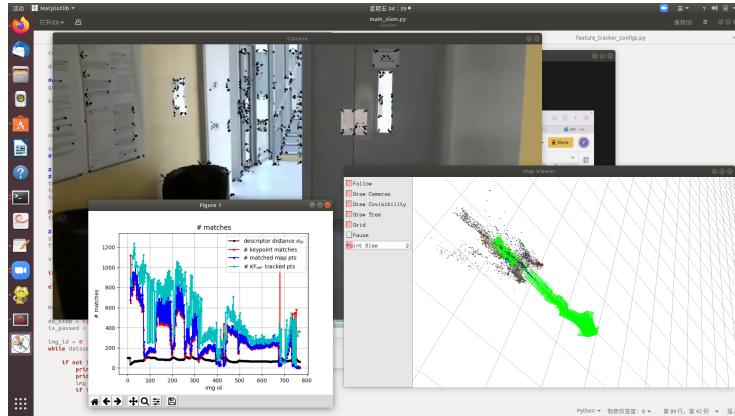


Figure 14: Les résultats de SHI-TOMASI comme détecteur et ORB comme descripteur

Dans le graphe de suivi d'appariement par la méthode SHI-TOMASI-ORB, on constate un niveau de 1000 en termes de points filtrés par le filtre de Kalman et une baisse significative le long de la trajectoire. De plus, on voit que le nombre des points appariés de la carte est toujours plus petit que celui de points filtrés. La distance en moyen entre les caractéristiques reste au niveau de 100, ce qui montre que le drone vole à une vitesse assez constante.

Il faut noter que le module PySLAM s'arrête au coin de tourner. Ce phénomène

est aussi observé au graphe d'appariement, où le nombre des points appariés de la carte tombe à 0 brutalement. En termes de carte 3D établie, on peut voir une forme de couloir droit avec beaucoup de bruit. Un suivi de trajectoire droite est aussi vue dans la carte 3D.

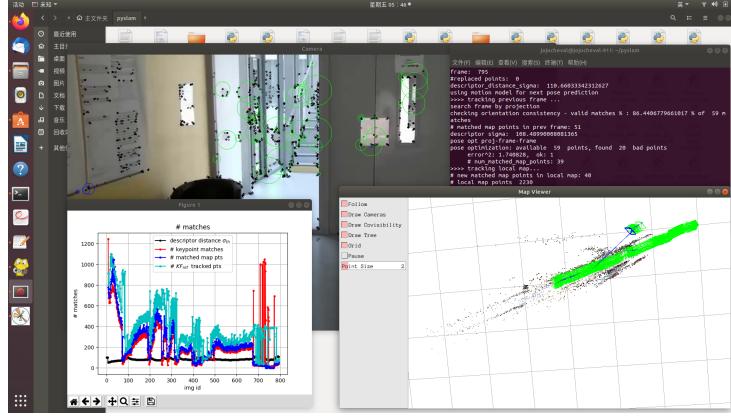


Figure 15: Les résultats de FAST comme détecteur et ORB comme descripteur

Le résultat de la méthode FAST-ORB se ressemble beaucoup à celui de SHITOMASI-ORB, ce qui est attendu car elles utilisent le même descripteur ORB. Il faut voir que la forme de couloir est plus claire et moins de points de bruits sont présentés dans la carte 3D établie par FAST-ORB.

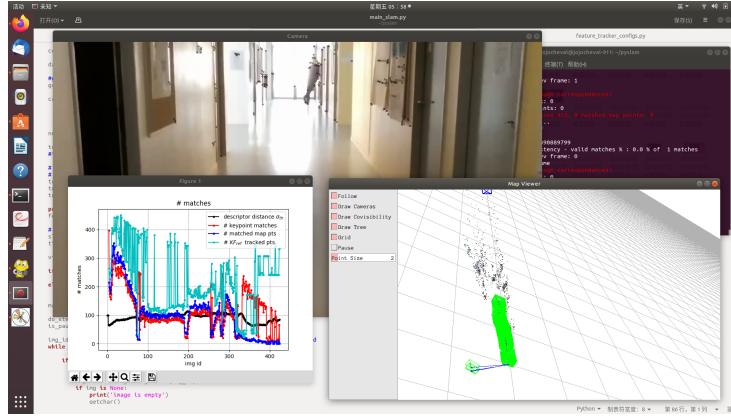


Figure 16: Les résultats de ORB comme détecteur et ORB comme descripteur

En revanche, le résultat est moins satisfaisant pour le cas où ORB comme détecteur et descripteur en même temps. On voit que peu de caractéristiques sont détectées (400 au début et 50 à la fin), ce qui limite beaucoup la qualité de carte établie. Cette insuffisance est aussi constatée dans la carte 3D où les points sont irréguliers.

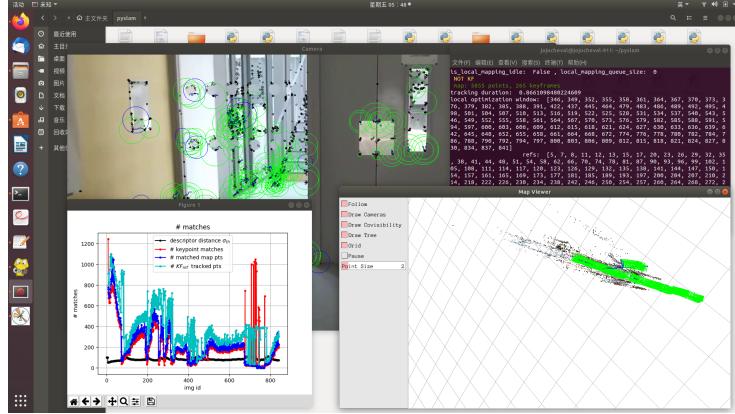


Figure 17: Les résultats de ORB2 comme détecteur et ORB2 comme descripteur

On procède au test de ORB2 comme détecteur et descripteur. Cette fois, on voit que le module PySLAM donne un meilleur résultat que ORB et le graphe d'appariement se ressemble beaucoup à celui de FAST-ORB. La forme de couloir est bien observée, tandis que l'étape de rotation échoue toujours. Même si l'on force l'algorithme à continuer, la position de drone après la rotation n'est plus cohérente avec sa trajectoire précédente, ce qui montre un échec d'estimation de position.

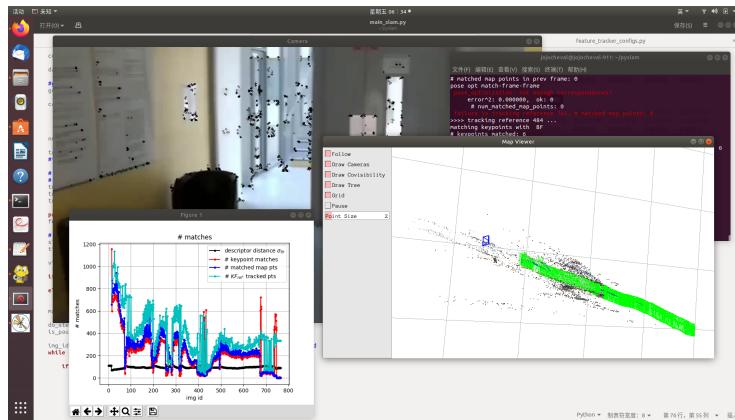


Figure 18: Les résultats de ORB2 comme détecteur et BEBLID comme descripteur

On va attaquer les cas où ORB2 sert comme détecteur. Dans le test de ORB2-BEBLID, le graphe d'appariement est pareil au cas précédents, mais plus de points de bruits sont constatés dans la carte 3D. Ce nuage de bruit est notamment présenté par deux rayons diverses de points, ce qui n'existe pas dans l'espace réelle.

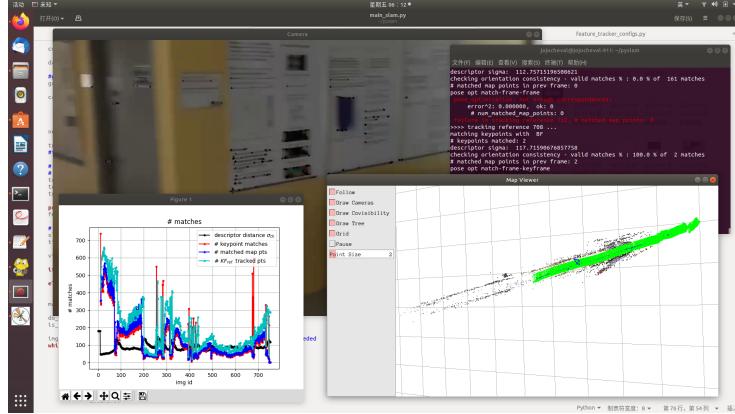


Figure 19: Les résultats de ORB2 comme détecteur et FREAK comme descripteur

Le résultat est plutôt satisfaisant dans la méthode de ORB2-FREAK. Dans le graphe d'appariement, on voit que moins de caractéristiques sont détectées (600 au début et 200 à la fin). En revanche, la carte 3D est beaucoup plus claire que celles des cas précédents. Cela montre que plus de caractéristiques détectées n'augmentent pas forcément la performance de l'algorithme et elles peuvent aussi produire plus de bruits.

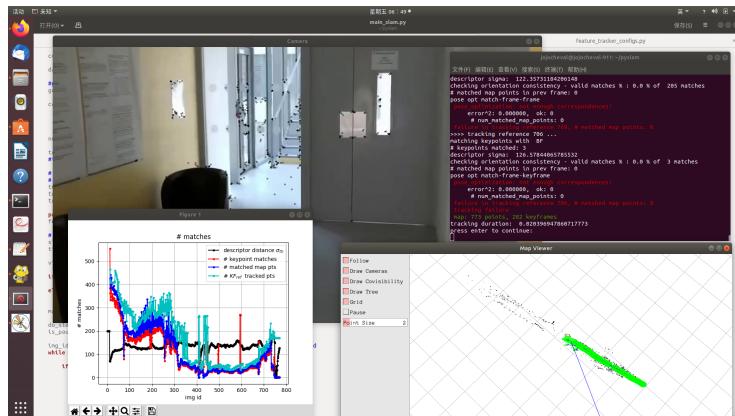


Figure 20: Les résultats de BRISK comme détecteur et BRISK comme descripteur

Cependant, le nombre de caractéristiques détectés est toujours un facteur important pour la qualité de carte établie. Dans le cas de BRISK comme détecteur et descripteur en même temps, on voit que seulement 400 caractéristiques sont détectées. Par conséquent, le nuage de points dans la carte est très rare, ce qui ne suffit pas à établir une carte utile.

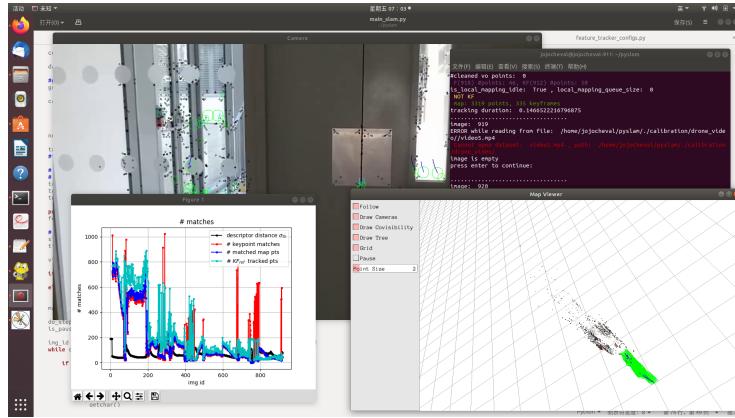


Figure 21: Les résultats de AKAZE comme détecteur et AKAZE comme descripteur

Le résultat est moins satisfaisant dans le cas de AKAZE. Lorsque le nombre des points appariés tombe brutalement vers 0 au milieu de test, le nuage de points ce casse aussi et on ne voit pas une forme de couloir.

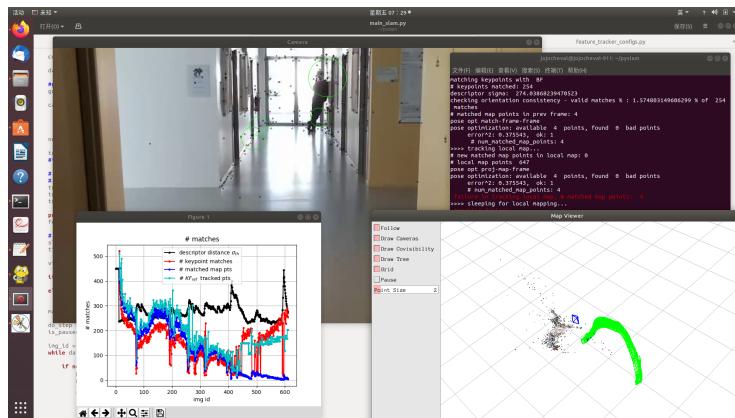


Figure 22: Les résultats de SIFT comme détecteur et SIFT comme descripteur

Le cas est même pire pour la méthode de SIFT. On voit une distance en moyen de 300 entre les caractéristiques, ce qui montre que l'appariement des caractéristiques ne fonctionne pas. Le nuage de points dans la carte est aussi insignifiant et la trajectoire traitée de la drone n'est plus droite.

Pour le cas de SUPERPOINT, le résultat est pareil. L'algorithme cherche à apparter les points de cartes, mais elle est limitée par le nombre de caractéristiques détectées. Par conséquent, la morphologie s'arrête avant que le drone tourne et la forme de couloir n'est pas vue. Il faut noter que la distance en moyen de

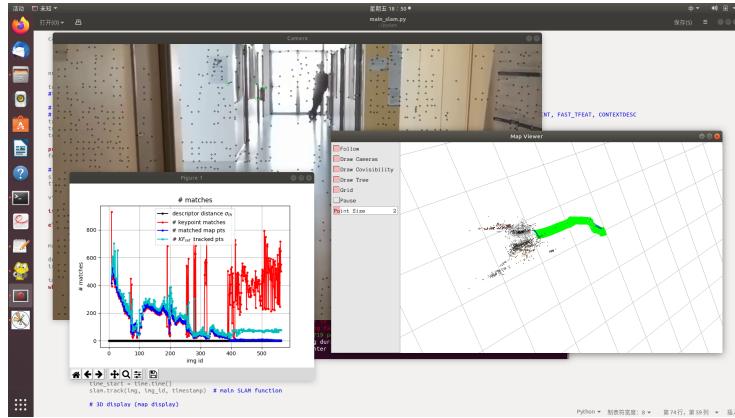


Figure 23: Les résultats de SUPERPOINT comme détecteur et SUPERPOINT comme descripteur

caractéristiques pour SUPERPOINT est 0, ce qui conduit au nombre limité de caractéristiques.

## 6.2 Arrêt fréquent de l'algorithme

On constate que le module PySLAM s'arrête souvent à cause d'échec d'appariement, soit entre les frames successives, soit entre les points caractéristiques et les points en 3D, ce qui consiste une autre raison que PySLAM ne peut pas être implémenté en temps réel.

L'origine de ces échecs d'appariement se trouve dans l'insuffisance des points caractéristiques. Puisque la SLAM monoculaire ne peut pas récupérer les profondeurs directement à partir des entrées comme lidars, son fonctionnement dépend beaucoup des détecteurs et des descripteurs de caractéristiques. Une fois que l'étape de détection manque son but, l'appariement sera aussi perdu. Par conséquent, le drone ne peut plus estimer sa position ainsi que sa vitesse dans la carte, il sera donc aveugle et la trajectoire sera aussi perdue. Dans le but de ne pas donner une carte erronée, le module PySLAM sera interrompu et un message d'erreur sera montré dans le terminal.

Plusieurs facteurs peuvent conduire à l'insuffisance des points caractéristiques. Premièrement, il peut arriver que l'image est toute plate comme une mur blanche et il manque des formes et des coins à être identifiés comme caractéristiques. Pour régler cette situation, on peut ajouter les objets dans le contexte pour que le drone puisse les détecter. Deuxièmement, l'insuffisance des caractéristiques peut être causée par l'insuffisance de la qualité d'image, par exemple, une image floue à cause de mouvement rapide. Pour cette cause, on propose d'essayer d'autres drones avec une meilleure caméra embarquée dans les projets suivants et de les contrôler le plus doucement possible pour obtenir une vidéo adaptée

à la SLAM monoculaire. Troisièmement, la puissance de détecteur et de descripteur peut aussi influencer les points de caractéristiques détectés, comme présenté dans la partie précédente. Il faut donc choisir une combinaison propre de détecteur et de descripteur pour obtenir un résultat satisfaisant.

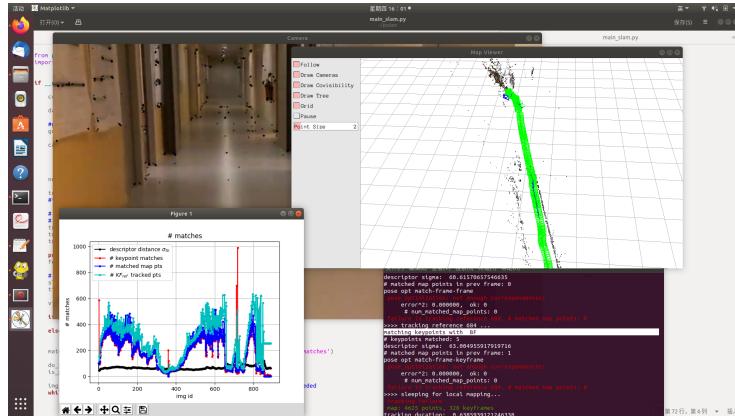


Figure 24: Exemple d'arrêt de l'algorithme

### 6.3 Difficulté de rotation

Une autre difficulté imprévue est observée quand le drone cherche à tourner au coin. Des messages d'erreur comme décrits dans la partie précédente sont envoyés au terminal. Par conséquent, le module de PySLAM s'arrête et il ne peut plus continuer même si on le force à traiter les frames suivant.

Une des causes de ce phénomène est liée à l'insuffisance des caractéristiques, comme présentée dans la partie précédente. Une autre raison peut être la vitesse trop rapide de rotation du drone, ce qui limite le nombre de frames pour traiter le processus de rotation et conduit finalement à l'échec d'appariement. Enfin, la cause la plus possible est que le drone n'avance pas en rotation. Le module de PySLAM a bien marché dans la vidéo de test enregistrée par une voiture, dont la trajectoire est un arc quand elle tourne au coin. Cependant, notre drone Tello a quatre hélices, ce qui lui permet de tourner sans avancer et c'est bien le cas dans notre vidéo de test. Par conséquent, la position de drone n'est pas mise à jour dans la carte et le module n'arrive pas à appairer les points en 3D existant et les nouvelles caractéristiques détectées dans la vidéo.

Ce problème de rotation n'a pas été résolu dans notre projet à cause du temps limité. Pour les projets au futur, on propose de faire intervenir la position réelle de drone, enregistrée par ROS en temps réel. Dans le module original de PySLAM, la position de caméra n'est pas connue et elle est calculée et estimée à partir de l'appariement de caractéristiques entre frames. Si on peut saisir sa position en temps réel, on peut donc fournir une référence au module

de PySLAM pour corriger la carte établie. Cette étape est particulièrement utile pour résoudre le problème de rotation, puisque l'on peut limiter les angles de recherches au traitement des frames de rotation afin d'augmenter la chance de réussie. Vu que le module PySLAM ne marche pas en temps réel, on propose d'enregistrer une liste de position de drone Tello avec horodatage et cette information peut être utilisée par le traitement hors-ligne.

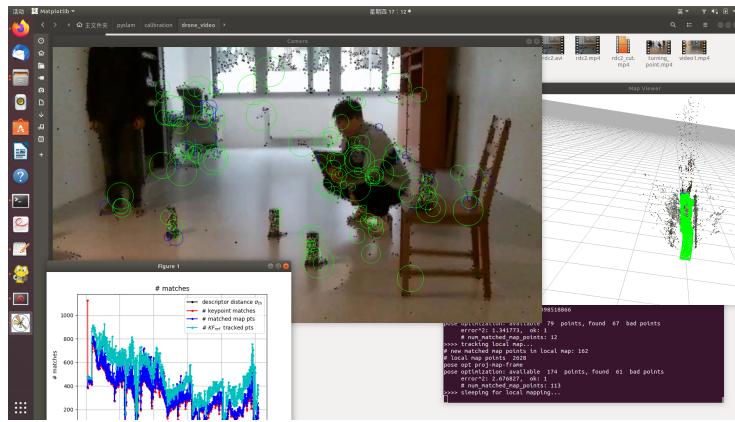


Figure 25: Situation de tourner dans la vidéo de test enregistrée par le drone

## 7 Conclusion

Dans ce projet, on a réalisé la communication et le contrôle entre ROS et Tello, et réalisé la localisation et la cartographie hors ligne via la bibliothèque open source PySLAM. En plus, on a essayé de résoudre le problème de la perte de points caractéristiques du drone lors d'un tournant et on a aussi comparé les résultats de différents sous-algorithmes de descripteurs et de détecteurs.

De plus, on a également rencontré des difficultés qui sont:

- SLAM n'est pas en temps réel en raison des exigences élevées en matière de puissance de calcul.
- Les instructions de contrôle du drone et l'enregistrement vidéo doivent être exécutés séparément en deux threads. Sinon il y aura des conflits et le drone ne pourra pas être contrôlé.
- Il y a de difficulté à tourner.
- La drone a des limitations de batterie, de perturbation de vol et de SLAM interne.
- Il y a un phénomène que le nuage de points 3D n'est pas aligné.

Pour nos Perspectives, On pense qu'on doit faire ces qui suits pour améliorer encore la performance.

- On peut optimiser les algorithmes pour l'intégrer dans le système ROS a permis d'accomplir la tâche du slam en temps réel.
- On peut faire une projection de nuage de points 2D.
- Il faudrait aussi améliorer la vidéo de test : + caractéristiques, - vitesse de rotation, - perturbation du vol.

## 8 Usage

Tous les codes de ce projet peut être trouvé dans ce lien de dépôt github :  
<https://github.com/Rescon/ROB314>.

### 8.1 La communication de TELLO via ordinateur

Dans le répertoire *DJITelloPy*, nous devons d'abord installer cette bibliothèque via *pip3* (comme indiqué dans le code ci-dessous).

```
1 pip3 install djitellopy
```

Ensuite, nous exécutons directement le fichier *manual-control-record-video.py* pour contrôler le drone TELLO et enregistrer son flux vidéo au format avi (l'ordinateur doit être connecté au Wi-Fi du drone).

```
1 python3 manual-control-record-video.py
```

### 8.2 La communication de TELLO via ROS

Dans le répertoire *drone\_racing\_ws*, exécutez la commande suivante pour vous connecter au drone TELLO (l'ordinateur doit être connecté au Wi-Fi du drone).

```
1 roslaunch tello_driver tello_node.launch tello_ip:="192.168.10.1"
```

Exécutez la commande suivante pour commander au drone TELLO en utilisant *teleop\_twist\_keyboard*.

```
1 rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/tello/cmd_vel
```

Exécutez la commande suivante pour enregistrer le flux vidéo provenant du drone TELLO en tant que fichier *output.avi*. en utilisant *video\_recorder*.

```
1 rosrun image_view video_recorder image:=/tello/image_raw _image_transport:=h264
```

### 8.3 SLAM

Afin d'utiliser cette bibliothèque open source, nous allons effectuer les étapes suivantes.

- Étape 1: Calibrer la caméra du drone TELLO sous le dossier **calibration**.

- Étape 2: Sous la bibliothèque open source PySLAM, on doit ajouter les paramètres de fonctionnement de la caméra au fichier de configuration *WEBCAM.yaml* sous le fichier de **setting**.
- Étape 3: Dans le fichier config.ini, on doit ajouter l'adresse de la vidéo et la *WEBCAM.yaml* modifiée.
- Étape 4: On peut modifier la détecteur et la descripteur dans le fichier **main\_slam.py**.
- Étape 5: Exécutez la commande suivante pour démarrer le positionnement et construire la carte hors ligne de la caméra.

```
1      python3 -O main_slam.py
```

## References

- Fritsch, J., Kuehnl, T., and Geiger, A. (2013). A new performance measure and evaluation benchmark for road detection algorithms. In *International Conference on Intelligent Transportation Systems (ITSC)*.