RAPPORT PROJET 5TC Option Robotique-Embarqué - 2017

Projet Crazyflie

Étudiants:
Alexandre CLAUDE, Département TC
Nathanael LEUTE, Département TC
David LOUP, Département TC

2016-2017

Enseignants suiveurs:
Stéphane d'ALu, Laboratoire CITI
Olivier Simonin, Laboratoire CITI

Table des matières

1 Introduction **2**

1.1 Objectif du projet . . . 2

1.2 Répartition du travail . . . 2

1.3 Les principales étapes du projet . . . 2

1.3.1 Création de l'environnement de travail . . . 2

1.3.2 Prise en main du matériel . . . 2

1.3.3 Détection avec la Kinect . . . 2

1.3.4 Contrôle du Crazyflie par la Kinect . . . 3

2 Matériel utilisé **3**

2.1 Le Crazyflie . . . 3

2.2 La Kinect . . . 4

3 Comportements robotiques et prise de décision **4**

3.1 Présentation de la stratégie de comportement . . . 4

3.1.1 Contrôle proportionnel seulement en z . . . 4

3.1.2 Ajout du contrôle en x . . . 4

3.1.3 Contrôle proportionnel sur les 3 axes . . . 4

3.1.4 L'algorithme final . . . 5

3.1.5 Tentatives non incluses dans le code final . . . 5

3.2 Développements logiciels et techniques . . . 5

3.2.1 Premières commandes du Crazyflie . . . 5

3.2.2 Codage de l'algorithme : Les fonctions tests . . . 6

3.2.3 Codage de l'algorithme : Le main et le callback . . . 7

3.2.4 Codage de l'algorithme : Les fonctions de calcul du PID . . . 9

3.2.5 Traitement d'image . . . 9

4 Expérimentations et validations **10**

4.1 Protocole expérimental . . . 10

4.1.1 La découverte de la réponse du Crazyflie . . . 10

4.1.2 L'acquisition avec la Kinect . . . 10

4.1.3 La mise en commun contrôle . . . 10

4.2 Résultats expérimentaux . . . 10

4.2.1 Détection du drone via la Kinect . . . 10

4.2.2 Correction de la position du drone . . . 11

5 Conclusion **12**

1 Introduction

1.1 Objectif du projet

L'objectif de ce projet est de repérer la position dans l'espace du drone Crazyflie à l'aide d'une caméra en profondeur 'Kinect' afin de lui faire réaliser un vol stationnaire autour d'un point de l'espace. Le drone sera commandé automatiquement (pas de pilotage manuel) via le framework ROS vu en cours. Le projet peut donc être séparé en 2 objectifs principaux : détecter la position du drone avec la Kinect, et en déduire les commandes adaptées pour rester autour du point voulu.

1.2 Répartition du travail

Pour mener à bien le projet, nous l'avons divisé, comme dit plus haut, en 2 tâches principales :

- Nathanaël travaillait sur la partie Kinect, c'est-à-dire la détection de la position du Crazyflie. Pour cela, il fallait :
  - Récupérer les informations de la Kinect avec ROS
  - Traiter cette information pour détecter le Crazyflie
  - Estimer la différence entre la position actuelle du Crazyflie et la position souhaitée
  - Renvoyer le résultat des traitements via ROS
- Alexandre et David étaient chargés de mettre en place le pilotage du drone avec ROS. Cela signifie :
  - Mettre en place l'environnement de travail : Création d'une VM intégrant les drivers du Crazyfile et où ROS est installé.
  - Faire des essais pour découvrir le pilotage et le comportement du drone avec ROS
  - Écrire le programme permettant d'utiliser l'écart entre la position du Crazyflie et la cible, obtenue grâce au travail de Nathanaël, dans le but de calculer et d'envoyer les commandes adéquates au drone pour qu'il reste le plus près possible de son but.

1.3 Les principales étapes du projet

Cette section présente les principales étapes du projet. Les détails techniques, tels que le code source, de ces étapes seront donnés dans la section **Développements logiciels et techniques**

1.3.1 Création de l'environnement de travail

Sur le site de Bitcraze, une VM est disponible. Elle contient une distribution Ubuntu avec les pilotes du dongle USB permettant de communiquer avec le Crazyflie. Il fallait donc installer ROS sur cette VM. La version de ROS utilisée sur le PC relié à la Kinect n'était pas compatible avec la version d'Ubuntu installée sur cette VM. Nous nous rendrons compte par la suite que cette différence de version n'est pas gênante dans notre projet.

1.3.2 Prise en main du matériel

Sur GitHub, un paquet ROS pour le contrôle du Crazyflie est disponible. Il fallait, à travers la documentation et les exemples fournis, découvrir comment fonctionnait ce paquet pour communiquer avec le Crazyflie. Nous pouvions ainsi effectuer nos premiers tests.

1.3.3 Détection avec la Kinect

Notre première approche consistait à récupérer l'image couleur de deux Kinects orthogonales. Le drone disposant de LEDs nous avons pensé qu'il serait possible de filtrer les images RGB provenant des Kinects afin d'isoler le drone dans chaque image et donc de le localiser dans l'espace. Cependant, cette approche n'a pas porté ses fruits. En effet, il nous était impossible de trouver les bons seuils permettant une détection parfaite du drone. Il y avait beaucoup de bruit sur l'image même après filtrage. C'est

pourquoi nous sommes passés à une approche image en profondeur acquise par une seule Kinect. Nous nous affranchissons ainsi des problèmes de bruits. Néanmoins du fait de son architecture matérielle, le drone avait tendance à disparaître lorsqu'il était trop éloigné de la Kinect. Ce problème a été résolu simplement en ajoutant du scotch autour du drone afin d'augmenter sa surface détectable.

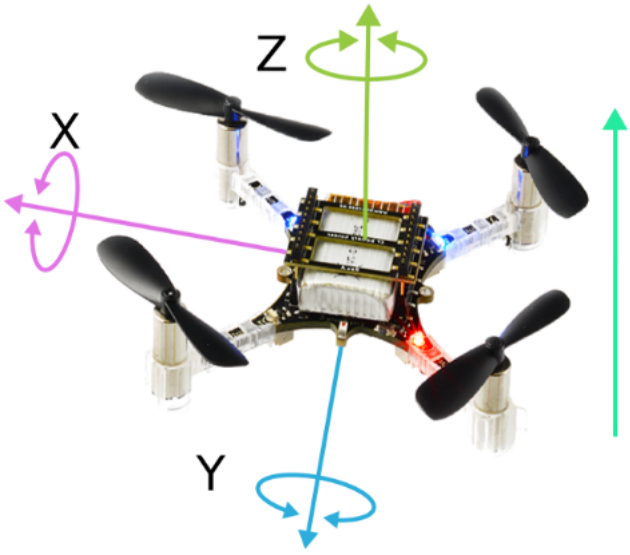
1.3.4 Contrôle du Crazyflie par la Kinect

Après avoir compris comment fonctionnait le paquet ROS et après être parvenu à obtenir sa position avec la Kinect, nous avons dû faire l'algorithme pour stabiliser le drone à l'endroit voulu. Cela s'est fait par de nombreux tests, afin d'ajuster les différents paramètres mis en place dans le programme de contrôle, détaillé plus loin. Nous avons notamment découvert à travers ces tests, que pour un même ordre, le Crazfylie réagissait différemment en fonction de la charge de sa batterie.

2 Matériel utilisé

2.1 Le Crazyflie

Le Crazyflie est un quadrotor (drone à 4 moteurs).Il peut effectuer des rotations et des translations sur les axes de l'espace **x,y** et **z**. Sur le drone, ces 3 axes sont orientés comme indiqué dans la figure suivante :



Le Crazyflie embarque les capteurs suivants : Accéléromètres, gyroscope, magnétomètre et capteur de pression. Il pèse 27g et a une autonomie en vol d'environ 7 minutes.
Pour contrôler le Crazyflie, nous avons eu besoin du matériel suivant :

- Du Crazyflie, bien sûr. Comme la batterie s'épuise rapidement lors des tests, nous avons utilisé 2 Crazyflie, afin d'avoir toujours au moins 1 drone chargé.
- Du dongle USB permettant d'établir un lien radio avec le Crazyflie
- d'un PC faisant tourner la VM, notre environnement de travail. Pour cela, nous avons utilisé nos ordinateurs personnels, ce qui nous permettait de travailler à 2 en même temps sur la programmation du Crazyflie. Pour le travail collaboratif, nous avons mis en place un dépôt git sur GitHub [1].

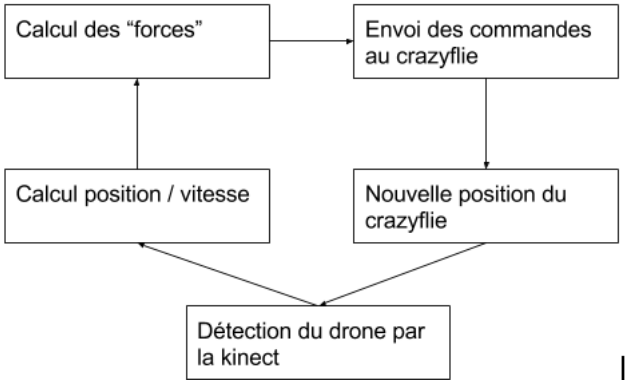
Une documentation technique du Crazyflie nous a également été fournie [2].

2.2 La Kinect

La partie détection du crazyflie a mobilisé le turtlebot **'NoName'** (avec une kinect surélevée) et le PC **Bacau** dont l'adresse IP était **192.168.0.101**. Le turtlebot n'a pas été utilisé en tant que tel. Nous nous en sommes servi pour sa kinect ainsi que son pc portable embarqué sur lequel ROS est installé. La kinect dispose d'une caméra RGB ainsi que de deux capteurs de profondeur 3D. Elle est très versatile et convient parfaitement pour notre tâche de localisation du drone.

3 Comportements robotiques et prise de décision

3.1 Présentation de la stratégie de comportement



L'algorithme utilisé pour le contrôle du drone a évolué au cours du projet. L'idée finale étant l'utilisation d'un contrôleur PID, censé réduire l'effet d'oscillation du drone autour de son but. Voici les différentes étapes du développement de l'algorithme de contrôle, entre le début et la fin du projet :

3.1.1 Contrôle proportionnel seulement en z

Nous avons commencé par contrôler simplement sur z (la hauteur) en indiquant un ordre de poussée sur les hélices proportionnelles à l'écart entre la hauteur du drone et la coordonnée en z de son but. Ainsi, plus le drone est loin du but, plus il met de la puissance pour arriver vers le but. Ce delta entre la position actuelle et le but était envoyé sur le topic /coord/pub par le PC du turtlebot relié à la Kinect. Il était ensuite récupéré par le nœud ROS qui contrôle le Crazyflie.

3.1.2 Ajout du contrôle en x

Étant donné que le drone n'est pas forcément équilibré sur les axes x et y. Nous nous sommes vite rendu compte que nous devions mettre en place le même mécanisme sur l'axe **x** afin d'éviter que le drone ne parte loin sur cet axe et sorte du champ de vision de la kinect. A ce stade, le Crazyflie commençait à osciller autour de son but. Il ne se stabilisait pas et dessinait de larges cercles autour de la position cible, tout en déviant petit à petit sur l'axe **y**, la profondeur, avant d'être trop loin pour être détecté par la Kinect. Les valeurs de poussée en z n'étant pas encore parfaitement réglées, il arrivait aussi souvent que le drone descende trop vite quand il se trouvait au-dessus du but, et n'avait pas assez de puissance pour remonter, il finissait donc par se poser, au lieu de remonter vers le but.

3.1.3 Contrôle proportionnel sur les 3 axes

Nous avons ensuite mis en place le même mécanisme sur les 3 axes. À ce stade, nous utilisons simplement la composante P du PID, en donnant un ordre **P**roportionnel à l'écart entre la position actuelle et

le but. Ce n'était pas suffisant pour stabiliser le drone autour de son but, le drone oscillait autour, avant de sortir de prendre trop d'élan et de sortir du champ de vision de la Kinect.

3.1.4 L'algorithme final

Nous avons donc mis en place un second coefficient du PID : la **D**érivée. avec la dérivé de la position du drone par rapport au temps, nous pouvons obtenir sa vitesse. Cette donnée doit nous permettre de contrôler la vitesse du drone en fonction de sa position. Par exemple :

- Si le drone est proche du but et arrive trop vite, il faut qu'il ralentisse, afin que son élan emmagasiné pendant son déplacement ne l'emmène pas au-delà du but.
- Au contraire, si le drone est loin, il peut augmenter sa vitesse pour arriver plus vite au but

Ce paramètre permet donc de réduire l'oscillation autour du but, en faisant ralentir le drone à l'approche du but. Nous avons donc changé le format des messages échangés entre le PC de la Kinect et le PC du Crazyflie pour y inclure la vitesse du drone.

3.1.5 Tentatives non incluses dans le code final

Nous avons exploré 2 pistes pour ce projet, avant de les abandonner, fautes de temps ou de résultats probants :

- Nous pensions que les données des capteurs d'accélération du Crazyflie pouvaient nous aider à le stabiliser. Ces données étaient disponibles sur le topic /**imu**. Les variations des valeurs envoyées par l'accéléromètre était trop faible pour en déduire une dérive du drone, sur tel ou tel axe : En claire, la déviation du Crazyflie était trop lente pour être perçue via les capteurs d'accélération. Nous nous sommes donc concentrés seulement sur les informations de la Kinect pour le contrôle du drone.
- L'intégration du dernier paramètre I (Intégrale) du PID, était censé permettre d'éviter les dérives lentes autour du but. N'ayant plus suffisamment de temps pour le comprendre et le tester en détail, nous l'avons finalement retiré de l'algorithme et nous sommes concentrés sur la dernière étape : le réglage des paramètres **P** et **D** au cours de multiples essais.

3.2 Développements logiciels et techniques

3.2.1 Premières commandes du Crazyflie

Pour commander le drone, nous passons par le paquet ROS pour le Crazyflie, disponible sur github [3].
La documentation du paquet est peu pourvue, mais elle se dote de quelques exemples fonctionnels à étudier. L'exemple *teleop_xbox360* permet de piloter le drone avec une manette de Xbox360, reliée au PC. En regardant le code source de cet exemple, nous avons compris comment lancer un noeud ROS Crazyflie pour lui envoyer des commandes de contrôle. Après s'être créé un workspace avec catkin et y avoir installé le paquet Crazyflie ROS, les commandes suivantes, tapées dans le workspace, permettent de lancer un noeud ros Crazyflie :

```
#Lance ROS et charge l'environnement catkin
>roscore
>source devel/setup.bash

# Lancer l'instance d'un serveur pour le Crazyflie
>roslaunch crazyflie_driver crazyflie_server.launch
# Cette commande renvoie une liste de lien radio disponible pour communiquer avec le drone :
>rosrun crazyflie_tools scan
uri:=radio:/10/100/80   #Exemple de valeur renvoyée
# Ajoute 1 drone à la liste de ceux pilotés par le serveur
>roslaunch crazyflie_driver crazyflie_add.launch uri:=radio:/10/100/80
```

```
>rosrun crazyflie_demo algo.py
#crazyflie_demo étant le nom de notre paquet ROS (créer avec cmake et catkin)
#algo.py le script python contenant l'algorithme de contrôle
```

Le code de l'algorithme de contrôle (ainsi que le code côté Kinect) est disponible sur notre dépôt git [1], dans le fichier **src/test_cmd/scripts/test_control_imu.py**.

Pour communiquer entre les noeuds ROS, nous pouvons utiliser un "topic" qui permet de diffuser des informations aux autres noeuds abonnés au même topic. Ainsi, le Crazyflie écoute le topic **/cmd_vel** pour recueillir ses ordres, le PC du Crazyflie écoute le topic **/coord/pub** dans lequel le PC de la Kinect publie les informations de positions et de vitesse du drone. Pour commander le Crazyflie, il nous faut des messages au format Twist. Ce format de message est composé de 2 vecteurs en 3 dimensions **x y z**. Le premier est le vecteur linéaire, qui permet d'effectuer des translations sur les 3 axes. Le deuxième est le vecteur de rotation, qui permet de faire des rotations autour des 3 axes. Pour faire décoller le Crazyflie, il faut donc envoyer un ordre au format "Twist" sur le topic **/cmd_vel** dans lequel on met une valeur sur la composante **z** du vecteur linéaire (translation vers le haut).
Ci-dessous, les éléments clés du code source sont détaillés :

3.2.2 Codage de l'algorithme : Les fonctions tests

```
#Monter Descendre
def test1(p):
        twist = Twist()

        r = rospy.Rate(10) #10 Hz
        twist.linear.z = 41000
        twist.angular.z = 0
        for i in range(0,15):
                print "monter"
                p.publish(twist)
                r.sleep()
        twist.linear.z = 39300
        for i in range(0,10):
                print "stable"
                p.publish(twist)
                r.sleep()
        twist.angular.z = 15
        for i in range(0,20):
                print "rotation"
                p.publish(twist)
                r.sleep()

        p.publish(twist)
        soft_land(twist,p,5)
```

Cette fonction **test1** est la fonction que nous utilisions pour réaliser nos premiers tests. En initialisant **twist.linear.z** à 41000 avant de publier le twist sur le topic /cmd_vel, nous commandons le Crazyflie pour qu'il décolle. La valeur 41000, ne fait pas décoller le drone à la même vitesse en fonction de la charge de sa batterie.
Grâce à ropsy.Rate et à l'instruction rospy.Rate.sleep, il est possible de contrôler la vitesse à laquelle une boucle s'exécute dans le code. On peut ainsi définir pendant combien de temps nous voulons envoyer cette ordre de monter. Dans le cas de **test1**, c'est 1.5 seconde, après quoi, on envoie une nouvelle valeur en **z** : 39300
Cette seconde valeur, 39300, est censée être la valeur d'équilibre de la force de poussée du drone. Avec cette valeur, le drone ne monte pas, et ne descend pas : il est stable en **z**. Durant nos tests, le drone se

stabilisait légèrement, mais finissait toujours, soit par monter petit à petit, soit par descendre, en fonction de la charge de la batterie.
On remarque qu'à la fin du **test1**, la fonction **solf_land** est appelée. Nous l'avons écrite pour faire un atterrissage progressif du drone, en lui donnant une valeur de poussée légèrement inférieure à sa valeur d'équilibre au lieu de simplement couper les moteurs.

```
#Translation
def test2(p):
        twist = Twist()
        rospy.Subscriber('imu', Imu, callback_sensor, twist)
        r = rospy.Rate(10) #10 Hz
        twist.angular.z = 0
        #stable en z
        twist.linear.z = 39300
        for i in range(0,20):
                print "stable"
                p.publish(twist)
                r.sleep()
        twist.linear.y = -30
        for i in range(0,20):
                print "stable"
                p.publish(twist)
                r.sleep()
        p.publish(twist)
        soft_land(twist,p,3)
```

Ci-dessus, le code de la fonction **test2**. Avec cette fonction, nous avons testé les commandes en translation du drone. Selon le même principe que la fonction **test1**, on utilise cette fois les composantes **x** et **y** du **Twist**. Nous avons laissé ces fonctions tests dans le code source, mais elles ne sont plus appelées dans le **main** du programme.

3.2.3 Codage de l'algorithme : Le main et le callback

Voici le code commenté du **main** de programme de contrôle du Crazyflie :

```
if __name__ == '__main__':
        #On initialise le noeud ROS
        rospy.init_node('crazyflie_test_controller', anonymous=True, log_level=rospy.WARN)
        twist = Twist()
        #Le noeud peut publier dans le topic cmd_vel, qui contrôle le Crazyflie
        p = rospy.Publisher('cmd_vel', Twist)
        global decollage
        global equilibre
        global max_pousse
        global min_pousse
        #Reglage du drone
        equilibre = 39500   #Valeur de poussée d'équilibre
        max_pousse = 42300  #La valeur maximum de poussée
        min_pousse = 39500  #La valeur minimum de poussée

        #Éteindre les LED, afin d'économiser de la batterie
        rospy.wait_for_service('update_params')
        update_params = rospy.ServiceProxy('update_params', UpdateParams)
        rospy.set_param("ring/effect", 0)
        update_params(["ring/effect"])
```

```
        decollage = True
        #On écoute le topic /coor/pub, dans lequel le PC Kinect va
        #publier les coordonnées et vitesse du drone.
        #A chaque nouveau message sur ce topic, on appel callback_kinect()
        listener = rospy.Subscriber('/coord/pub', String, callback_kinect, (twist))
        try:
                decollage = False
                print("Fin decollage")
                #Avant d'écouter les messages de la kinect, on met le drone à sa valeur d'équilibre
                twist.linear.z = equilibre
                #On publie le message sur le topic vmd_vel pour piloter le Crazyflie
                p.publish(twist)
                #La commande doit être publiée sans interruption pour que le drone continue de l'exé
                while not rospy.is_shutdown():
                        p.publish(twist)
        #arret urgence
        except KeyboardInterrupt:
                print("arret urgence")
                clear_twist(twist)
                p.publish(twist)
```

Le **main** se charge donc de l'initialisation du programme : équilibre du drone, inscription sur les topics nécessaires, définition des paramètres du drone (valeur min et max de la poussée). Les valeurs de poussées à envoyer sont calculées selon le principe du contrôleur PID, comme expliqué dans la section **Présentation de la stratégie de comportement**. Les coefficient **Proportionnel** et **Dérivé** du PID sont modifiés dans la fonction de **callback** du topic **/coord/pub** appelée **callback_kinect** dont le code est présenté ci-dessous :

```
#Cette fonction de callback est appelée a chaque fois qu'un message est reçu sur le
#topic /coord/pub, sur lequel le PC Kinect publie l'écart entre la position du drone
#et son but ainsi que la vitesse du Crazyflie.
def callback_kinect(data, args):
        twist = args
        global decollage
        # Le message reçu est parsé afin d'obtenir les coordonnées en x,y,z du but
        # ainsi que la vitesse du drone, toujours sur les 3 axes.
        target_pos = str(data).split(" ")
        x = target_pos[1]
        y = target_pos[2]
        z = target_pos[3]
        current_x_speed = target_pos[4]
        current_y_speed = target_pos[5]
        current_z_speed = target_pos[6]
        #On affiche un message de debug
        print("x:"+x+" y:"+y+" z:"+z)
        if int(decollage) == False:
                #Si la phase de décollage est terminée,
                #on calcul les paramètres du twist à envoyer au drone,
                #en appelant les fonction calc_z, calc_y et calc_x
                twist.linear.z = calc_z(z, current_z_speed, twist.linear.z)
                twist.linear.y = calc_y(y, current_y_speed, twist.linear.y)
                twist.linear.x = calc_x(x, current_x_speed, twist.linear.x)
        else:
                print("Mode decollage : Delta non pris en compte")
```

3.2.4 Codage de l'algorithme : Les fonctions de calcul du PID

La fonction **callback_kinect** appel donc les fonctions **calc_z**, **calc_y** et **calc_x** pour calculer l'ordre a donner. Voici leurs code source commenté :

```
def calc_z(delta, current_z_speed, current_z):
        #Valeur de P et D du PID, trouvée au cours des tests
        p = -13 #Quand la coordonnée en z est négative, on est en dessous du but.
        #Il faut donc un coefficient négatif pour augmenter la poussé quand
        #on est en dessous du but et la diminuer quand on est au dessus.
        d = 20
        global equilibre
        global max_pousse
        global min_pousse

        #Si on est au dessus de la cible, on se met à la valeur de poussée minimum
        #pour descendre vers le but, pas d'utilisation du PID dans ce cas.
        if int(delta) > 0:
                current_z = min_pousse
                res = current_z
        else:
        #Si on est en dessous de la cible, on utilise le PID :
        #delta représente l'écart en z entre le drone et le but
                res = int(current_z + p*int(delta) + d*float(current_z_speed))

        #Borne min et max
        if res > max_pousse:
                res = max_pousse
        elif res < min_pousse:
                res = min_pousse
        #Le drone n est plus dans le champs de vision de la kinect, il faut descendre.
        #-230 est l'écart entre le but et le sol. Si on obtient cette valeur, cela signifie
        #que le drone est hors du champs de vision de la Kinect.
        if int(delta) < -230:
                res = equilibre

        print("old z: \%s,  new z: \%s = \%s + \%s + \%s" \% (str(current_z),
            str(res), str(current_z),str(p*int(delta)), str(d*float(current_z_speed))))
        return res
```

Le code des 2 autres fonctions **calc_x** et **calc_y** est disponible sur le git du projet. Il a la même structure que la fonction **calc_z**, et utilise le même principe. Les coefficients du PID sont propres à chaque fonction, car le PID "idéal" est différent sur les 3 axes. Les valeurs de P et D ont été trouvées au cours des tests, en analysant les messages de débug pour voir si le coefficient était trop, ou pas assez influent sur les valeurs finales de poussés et de translations.

3.2.5 Traitement d'image

Le traitement d'image a été réalisé en Python. Nous récupérons l'image en profondeur à l'aide de la librairie **OpenCV**. Cette même libraire dispose d'une fonction renvoyant les coordonnées de l'élément le plus proche dans l'image. La localisation du drone est donc relativement simple : il s'agit du point le plus proche détecté par la Kinect. Une fois les coordonnées brutes récupérées nous effectuons différentes opérations afin d'obtenir le décalage par rapport au but et la vitesse du drone. Ces données sont ensuite publiées dans le topic **/coord/pub**.
Ce traitement est réalisé en continu, directement depuis le pc turtlebot. En effet, l'image en profondeur est représentée par une matrice 640x480. Le fait de publier ces images sur le réseau sans-fil local turtlebot

saturait le canal. On ne pouvait recevoir qu'une image toutes les 0.5 seconde.
L'envoi de commande réactive était donc impossible. C'est pourquoi le programme de détection du drone est exécuté directement sur le pc turtlebot.

4 Expérimentations et validations

4.1 Protocole expérimental

Le protocole expérimental a été découpé en trois étapes :

4.1.1 La découverte de la réponse du Crazyflie

Nous commencions donc nos expériences en faisant décoller le drone afin de comprendre comment il réagissait aux commandes et quels étaient les ordres de grandeur des valeurs de poussées et de translations sur chaque axe. Ces tests nous on permit de connaître la force à mettre en z pour être proche de l'équilibre en altitude.

4.1.2 L'acquisition avec la Kinect

Pour localisé le drone, nous l'avons placé devant la Kinect afin de voir si il était suffisamment épais pour être détecté par la Kinect. Après avoir constaté que le drone était trop fin, du ruban adhésif blanc a été ajouté tout autour du Crazyflie afin de lui donner de l'épaisseur.
Avec ce procédé, non seulement la Kinect détectait bien le Crazyflie, mais ce dernier semblait plus stable, car il le ruban adhésif rajoutait du poids réparti de manière à peu près uniforme autour du drone. On estimait ensuite sa position comme étant celle de l'élément le plus proche vue par la caméra 3D.

4.1.3 La mise en commun contrôle

Une fois ces 2 parties réalisées indépendamment, puis mises en communs, nous procédions à de nouveaux tests :
Nous avons d'abord tenté de faire décoller le drone et de le stabiliser dans sa montée au niveau de la cible. Cette méthode fonctionnait parfois, mais bien trop souvent, le drone ne parvenait pas à perdre assez d'élan avant d'arriver au but, il passait alors au dessus et commençait à osciller autour trop amplement, il finissait par continuer sa monté jusqu'au plafond. En changeant les valeurs min et max pour donner moins de force au décollage du drone, l'inverse se produisait, il décollait trop peu et ne parvenait jamais à monter suffisamment haut pour atteindre le but.
Au bout de nombreux tests, nous avons conclu que nous ne maîtrisions pas assez la phase de décollage. Plutôt que de le faire décoller en le faisant passer d'une phase de décollage à une phase de stabilisation, nous avons décidé de faire nos tests en lâchant le drone directement dans le champ de vision de la Kinect, comme on peut le voir sur les vidéos de démonstrations [4]. Ainsi, sa valeur de poussée en z est directement une valeur proche de son équilibre, ce qui facilite sa stabilisation, car il commence le vol avec peu d'inertie.

4.2 Résultats expérimentaux

4.2.1 Détection du drone via la Kinect

Comme évoqué plus haut, la détection du drone a été couronnée de succès, nous arrivons bien à détecter l'écart entre la position souhaitée et la position réelle du drone, ainsi que sa vitesse. Il faut cependant garder à l'esprit que la Kinect a une aire de détection optimale située entre 30cm et 1m. De ce fait, si jamais le drone s'écarte trop, ou s'il avance trop, il sortira de cette zone de détection.

4.2.2 Correction de la position du drone

Nous sommes finalement parvenus à un résultat acceptable. La démonstration finale a été filmée, [4] nous pouvons voir que le drone oscille autour de son point cible. Cela montre que la boucle de contrôle détecte bien un écart et essaye de le contrebalancer en envoyant des ordres permettant de rectifier la position du drone.
Malheureusement, le drone s'écarte rapidement de sa position d'équilibre, et finit par sortir du champ de détection de la Kinect au bout d'une trentaine de secondes. Nous discuterons des raisons de ce problème dans la partie qui suit.

5 Conclusion

Les deux objectifs initiaux étaient :

— Détecter la position du drone avec la Kinect et publier ces données.

— En déduire les commandes adaptées pour rester autour du point voulu.

Le premier objectif a été atteint, le deuxième que partiellement, dans le sens où le drone ne reste qu'un laps de temps limité autour de son point d'équilibre et finis par sortir du champ de détection de la Kinect. Nous n'avons pas pu apporter de solutions à toutes les problématiques rencontrées, mais nous proposons de conclure sur les pistes envisageables pour résoudre ces problèmes et continuer le projet :

Le drone est très léger. De ce fait, il est par nature difficile de le manœuvrer avec précision. C'est en partie pourquoi nous avions du mal à contrôler son oscillation. Nous avions tenté, sans succès dans le temps imparti, d'utiliser les capteurs sur drone, mais peut-être qu'un travail approfondi dans cette direction permettrait d'aider à la stabilité du drone.

Ensuite, le Crazyflie est très sensible à son niveau de charge. Prenons un exemple concret : le cas du vol stationnaire. Pour que le drone reste à une altitude donnée, il faut lui donner une valeur de poussée en z. Cette valeur sera plus élevée si le niveau de charge de la batterie du drone est faible. Nous n'avons pas eu le temps de prendre en compte ce paramètre dans le calcul des commandes, mais la valeur de la charge de la batterie est publié dans un topic. Il est certainement possible d'utiliser cette donnée pour corriger les commandes du drone.

Enfin, nous n'avons pas intégré le paramètre I du PID,*l'Intégrale*, notre PID est donc certainement améliorable.

Examiner ces pistes serait donc un bon point de départ pour un travail futur sur ce sujet.

Références

[1] CLAUDE Alexandre, LEUTE Nathanael, and LOUP David. Dépot git du projet, Déc 2016. https ://github.com/loupdavid/crazyflie_project.

[2] Bitcraze. Documentation bitcraze pour le crazyflie 2.0, 2016. https ://wiki.bitcraze.io/projects :crazyflie2 :index.

[3] Whoeing. Package ros for crazyflie, Nov 2016. https ://github.com/whoenig/crazyflie_ros.

[4] CLAUDE Alexandre, LEUTE Nathanael, and LOUP David. Vidéo finale, Déc 2016. https ://drive.google.com/file/d/0BwjCfn9CiyduTERETTREdW1XS28/view.