Documentation Tello EDU

Baptiste BEHELLE

Septembre 2021

Table des matières

1 Driver et package utilisés

Pour communiquer avec le drone Tello EDU :
`https://github.com/appie-17/tello_driver.git`
`http://wiki.ros.org/tello_driver`

Pour récupérer les informations de la caméra :
`http://wiki.ros.org/camera_info_manager_py?distro=kinetic`

Pour faire la liaison avec OpenCV :
`http://wiki.ros.org/vision_opencv?distro=noetic`

Installation des packages à partir des sources dans l'environnement de travail catkin, s'il manque des paquets ou des dépendances, ils sont installables depuis les dépôts Linux.

2 Driver du Tello EDU

On commence par allumer le drone puis on se connecte sur le réseau wifi du drone. On peut alors lancer le node avec :

```
roslaunch tello_driver tello_node.launch
```

On a alors accès à plusieurs topics :

Publisher :

/tello/image_raw sensor_msgs/Image : Flux vidéo de la caméra au format natif d'image de ROS

/tello/imag/raw/h264 h264_image_transport/H264Packet : Flux vidéo de la caméra au format h264 (MPEG-4 AVC)

/tello/odom nav_msgs/Odometry : Donne la position du drone, son orientation et sa vitesse

/tello/imu sensor_msgs/Imu : Donne la vitesse du drone à partir de l'imu

/tello/status : Donne des infos sur le drone, temps de vols, niveau de batterie,etc

Subscriber :

/tello/cmd_vel geometry_msgs/Twist : Envoie d'une consigne de vitesse au drone

/tello/emergency std_msgs/Empty : Arrêt d'urgence où on coupe tous les moteurs

/tello/land std_msgs/Empty : Consigne d'atterrissage

/tello/takeoff std_msgs/Empty : Consigne de décollage

Concernant le format des messages, pour les commandes telles que land, emergency ou takeoff on envoie un message vide, la simple réception du message suffit pour que le driver envoie l'instruction voulue au drone.

Pour cmd_vel, on envoie un message au format suivant :

```
linear:
  x: 0.0
  y: 0.0
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.0
```

La vitesse linéaire est en m/s tandis que la vitesse angulaire est en rad/s. Le driver reçoit cette instruction qu'il traduit pour le Tello en appliquant une conversion et un ratio. Cependant, il y a peu de documentation sur ce ratio et ce qu'il implique.

Concernant la vitesse angulaire, l'axe x et y ne serve à rien, car le drone ne

peut pas changer d'inclinaison, il peut juste tourner sur lui-même.

On peut également voir le retour de la caméra depuis un autre terminal avec :

```
rosrun rqt_image_view rqt_image_view /tello/image_raw/h264
```

3 Node de contrôle au clavier

Création d'une node pour contrôler le drone au clavier, le mappage des commandes avec les touches est le suivant :

z,q,s,d : déplacement avant, gauche, arrière, droite
a,e : rotation gauche, droite
w,x : monter, descendre
+,- : augmenter, réduire la vitesse linéaire
c,v : augmenter réduire la vitesse angulaire
& : emergency, éteint tous les moteurs
é : takeoff, fait décoller le drone
¨ : land, fait atterrir le drone
Ctrl + C : coupe le programme

les autres touches ne font rien.

Le programme peut se découper en 3 parties, on initialise les différents topics, on récupère les entrées claviers et on fait l'action associée à la touche entrée.

Init des topics :

```python
def __init__(self):

    #speed
    self.speed = 0.2 # en m/s
    self.speed_ang = 0.2 # en rad/s

    # Start ROS node
    rospy.init_node('keyboard_teleop_node')

    # Load parameters

    self.pub_takeoff = rospy.Publisher(
        'takeoff', Empty,  queue_size=1, latch=False)

    self.pub_land = rospy.Publisher(
        'land', Empty,  queue_size=1, latch=False)

    self.pub_emergency = rospy.Publisher(
        'emergency', Empty,  queue_size=1, latch=False)

    self.pub_cmd_out = rospy.Publisher(
        'cmd_vel', Twist, queue_size=10, latch=False)

    rospy.loginfo('Keyboard teleop node initialized')
```

Récupération de l'entrée clavier :

```python
#aquisition de la touche clavier
key = ''

tty.setraw(sys.stdin.fileno())

rlist, _, _ = select.select([sys.stdin], [], [], 0.5)
if rlist:
    key = sys.stdin.read(1)

termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)

print(key)
```

Envoie d'une commande :

```
if key == 'z':
    cmd.linear.y = self.speed
    print("speed forward (y): ")
    print(cmd.linear.y)
```

```
self.pub_cmd_out.publish(cmd)
```

4 Node de contrôle automatique

Le but de ce programme était d'écrire des fonctions de déplacement linéaire et angulaire pour pouvoir les enchainer et programmer des vols automatiques.

Pour ce faire, on utilise les données du topic /tello/odom qui nous donne la position relative du drone depuis son décollage, il donne également les valeurs de vitesse (qu'on peut aussi retrouver dans le topic /tello/odom)
Le drone calcule sa position grâce à la caméra sous le drone qui permet de faire du visio positionning en regardant l'évolution de points au sol pour déterminer son déplacement.

Les messages d'odométrie sont sous la forme :

```
pose:
  pose:
        position:
        x: -0.015279734507203102
        y: 0.013825475238263607
        z: -0.0006212824955582619
        orientation:
        x: 0.13802675902843475
        y: 0.07457036525011063
        z: 0.7522124648094177
        w: 0.6399721503257751
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
        linear:
        x: 0.002
        y: 0.002
        z: -0.005
        angular:
        x: 0.25908946990966797
        y: 0.4113595485687256
```

```
        z: -0.09298528730869293
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
  0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

C'est la partie pose qui nous intéresse pour notre algorithme.

On utilise les mêmes techniques que précédemment pour publier les commandes, on les fait juste boucler pour atteindre l'objectif de distance et de direction voulue.

Juste avant de donner un exemple du code pour une direction, il me reste 2 précisions à donner :
Les axes de commandes du drone ne sont pas les mêmes que ce qui proviennent des données d'odométrie :

```
repere odom :
 x (avant)
 y (droite)
 z (haut)
repere cmd_vel :
 x(droite)
 y (avant)
 z (haut)
```

De plus, les valeurs d'angles dans l'odométrie sont au format :

```
orientation:
        x: 0.13802675902843475
        y: 0.07457036525011063
        z: 0.7522124648094177
        w: 0.6399721503257751
```

Il s'agit d'une valeur en quaternion, il faut donc les traduire en valeur de radian, voir même d'angle pour avoir un affichage plus exploitable.

Exemple de code pour se déplacer vers une direction sur une distance voulue :
On récupère les messages d'odométrie pour avoir la position :

```
    #changer ici + les axes si utilisation d'un autres topics
    self.sub_odom = rospy.Subscriber(
        '/tello/odom', Odometry, self.calculate_pos)
    rospy.loginfo('autopilot node initialized')
    time.sleep(1)

#récupère la position du drone de façon relative à partir des messages d'odométrie
#attention c'est une valeur approché risque d'erreur d'où la nécessité d'une solution de positionnement absolu
def calculate_pos(self, msg):
    self.last_pos = msg.pose.pose.position
    self.last_ori = msg.pose.pose.orientation
```

La boucle qui tourne jusqu'à ce qu'on ait parcouru la bonne distance :

```
if direction == "forward":
    pos_ini_x = self.last_pos.x
    pos_ini_y = self.last_pos.y
    reste = -distance

    while reste > precision_lin or reste < -precision_lin:
        if reste < 0:
            self.move("forward")
        else:
            self.move("back")
        #si systeme parfait sans perturbation (car sinon le deuxieme terme qui n'est pas sensé varié créerai de l'erreur)
        reste = abs(self.last_pos.x-pos_ini_x) + abs(self.last_pos.y - pos_ini_y) - distance
        print("deplacement restant: "+str(reste))
        print("~~~")
```

Cette fonction est imparfaite, car elle considère que le déplacement est parfait sans perturbation. S'il y en avait, la valeur de déplacement serait incorrecte, car ici il pourrait y avoir du déplacement latéral pris en compte dans le calcul et donc on parcourrait moins que la distance voulue.
D'où la nécessité d'un positionnement absolu pour s'affranchir de ce problème.

5 Node d'analyse d'image avec OpenCV

Pour traiter le flux vidéo pour des futurs algorithmes de deep learning ou de tracking par exemple, il fallait préparer des outils pour traiter l'image.

Pour cela, on va utiliser la librairie OpenCV. Il faut donc l'interfacer avec ROS d'où l'utilisation du package cv_bridge qui va réaliser l'interface entre les deux. Il permet de convertir le flux vidéo d'un topic envoyant des messages Sensor_msgs/Images dans le format d'OpenCV ainsi que l'opération inverse pour publier.

Il nous faut donc un flux au format Sensor_msgs/Images et pour cela avec ce driver du Tello on a deux possibilités :
De base, le driver envoie un flux au format h264, on ne peut donc pas l'exploiter telle quelle mais, grâce à la commande suivante on peut le capter et le republier dans le format souhaité ici en Sensor_msgs/Images :

```
rosrun image_transport republish h264 in:=/tello/image_raw raw
out:=/h264toraw/image_raw _image_transport:=h264
```

Ou on change dans les paramètres du driver pour l'envoyer par défaut en Sensor_msgs/Images
Pour cette première application, j'ai réalisé un lecteur de qr code et de code barre à partir des images converties et de la librairie pyzbar qui décode les qr codes présents dans l'image.
Le code se décompose alors en 3 parties :
À chaque frame capté sur le topic on les traduit au format d'OpenCV :

```
#formatage de l'image du format image de ros vers un format exploitable par opencv
try:
    cv_image = self.bridge.imgmsg_to_cv2(data, "bgr8")
    print("img convert")
except CvBridgeError as e:
    print(e)
```

Analyse de l'image pour traduire les qr code :

```
#fonction qui repère et traduit un qr code ou un code barre dans la frame
def code_decoder(img):
    for barcode in decode(img):
        data = barcode.data.decode('utf-8')
        typecode = barcode.type
        print("code type : " + typecode)
        print("code lu : " + data)
        pts = np.array([barcode.polygon],np.int32)
        pts = pts.reshape((-1,1,2))
        cv2.polylines(img,[pts],True,(255,0,255),5)
        pts2 = barcode.rect
        cv2.putText(img,typecode+" : "+data,(pts2[0],pts2[1]),cv2.FONT_HERSHEY_SIMPLEX,0.9,(255,0,255),2)
    return img
```

Reconversion de l'image et publication sur un topic :

```
#reformatage et envoi sur le topic de publication
try:
    self.image_pub.publish(self.bridge.cv2_to_imgmsg(cv_image, "bgr8"))
    print("img send")
except CvBridgeError as e:
    print(e)
print("#####")
```

On peut alors afficher le résultat en utilisant la console de ROS pour afficher le flux de sortie avec :

```
rosrun rqt_image_view rqt_image_view /opencv_ros/img
```

6 Deep learning à partir d'un réseau déjà entrainé

Pour cette partie, il faut d'abord installer tensorflow et télecharger les drivers graphiques nvidia indiqués sur leur site pour faire fonctionner le mode gpu.

Pour le package, il contient dans le dossier source les codes nécessaires au fonctionnement du mode object detection de tensorflow, ils sont trouvables sur le github officiel de tensorflow.

Pour le code :
Chargement du modèle d'analyse de l'image :

```
##################################RECUP PARAM#########################################################################

node_name = str(rospy.get_param('node_name'))
model_name = str(rospy.get_param('model_path'))
label_name = str(rospy.get_param('label_path'))
topic_src = str(rospy.get_param('topic_src'))
topic_dest = str(rospy.get_param('topic_dest'))

##################################INIT TENSORFLOW + MODEL#############################################################

# patch tf1 into `utils.ops`
utils_ops.tf = tf.compat.v1

# Patch the location of gfile
tf.gfile = tf.io.gfile


model_path = os.path.join(os.path.dirname(sys.path[0]),'data','models', model_name,'saved_model')
label_path = os.path.join(os.path.dirname(sys.path[0]),'data','labels',label_name)

detection_model = tf.saved_model.load(model_path)
category_index = label_map_util.create_category_index_from_labelmap(label_path, use_display_name=True)
```

Fonction qui analyse l'image et trouve les différents objets :

```python
def run_inference_for_single_image(model, image):
    image = np.asarray(image)
    # The input needs to be a tensor, convert it using `tf.convert_to_tensor`.
    input_tensor = tf.convert_to_tensor(image)
    # The model expects a batch of images, so add an axis with `tf.newaxis`.
    input_tensor = input_tensor[tf.newaxis,...]

    # Run inference
    output_dict = model(input_tensor)

    # All outputs are batches tensors.
    # Convert to numpy arrays, and take index [0] to remove the batch dimension.
    # We're only interested in the first num_detections.
    num_detections = int(output_dict.pop('num_detections'))
    output_dict = {key: value[0, :num_detections].numpy()
                   for key, value in output_dict.items()}
    output_dict['num_detections'] = num_detections

    # detection_classes should be ints.
    output_dict['detection_classes'] = output_dict['detection_classes'].astype(np.int64)

    # Handle models with masks:
    if 'detection_masks' in output_dict:
        # Reframe the the bbox mask to the image size.
        detection_masks_reframed = utils_ops.reframe_box_masks_to_image_masks(
                                    output_dict['detection_masks'], output_dict['detection_boxes'],
                                    image.shape[0], image.shape[1])
        detection_masks_reframed = tf.cast(detection_masks_reframed > 0.5, tf.uint8)
        output_dict['detection_masks_reframed'] = detection_masks_reframed.numpy()

    return output_dict
```

Puis on modifie l'image d'origine pour rajouter les boxes :

```python
#analyse de l'image
output_dict = run_inference_for_single_image(detection_model,image_in)

#modification de l'image avec les resultats de l'analyse
vis_util.visualize_boxes_and_labels_on_image_array(
    image_modif,
    output_dict['detection_boxes'],
    output_dict['detection_classes'],
    output_dict['detection_scores'],
    category_index,
    instance_masks=output_dict.get('detection_masks_reframed', None),
    use_normalized_coordinates=True,
    line_thickness=8)
```

Et on republie l'image modifiée

Voici le launch sur les différents arguments à ajouter :

```
<launch>

    <arg name="node_name" default=""/>
    <arg name="model_path" default=""/>
    <arg name="label_path" default=""/>
    <arg name="topic_src" default=""/>
    <arg name="topic_dest" default=""/>

    <param name="node_name" value="$(arg node_name)" />
    <param name="model_path" value="$(arg model_name)" />
    <param name="label_path" value="$(arg label_name)" />
    <param name="topic_src"  value="$(arg topic_src)"/>
    <param name="topic_dest"  value="$(arg topic_dest)"/>

    <node pkg= "object_detector" name="$(arg node_name)" type="detect_ros.py"  output="screen"/>
</launch>
```

7 Commande pour faire fonctionner les différents programmes

Roscore

On allume le Tello et on se connecte sur son point d acces wifi

Roslaunch tello_driver tello_node.launch

Roslaunch tello_driver key_teleop.launch

Roslaunch opencv_ros opencv_ros_node.launch node_name:="opencv_ros" topic_src:="/h264toraw/image_raw" topic_dest:="/opencv_ros/img"

Roslaunch object_detector object_detector.launch node_name:="detector_object" model_name:="model_faster_rcnn_resnet_custom" label_name:="labelmap.pbtxt" topic_src:="/image" topic_dest:="detector_object/image"

Rosrun image_transport republish h264 in:=/tello/image_raw raw out:=/h264toraw/image_raw _image_transport:=h264

Rosrun rqt_image_view rqt_image_view