Person detection, tracking, and estimation using the onboard camera of a drone (aerial robots)

Monday 20th June, 2022 - 21:05

Rafidison Santatra Rakotondrasoa
University of Luxembourg
Email: rafidison.rakotondrasoa.001@student.uni.lu

Jose Luis Sanchez Lopez
University of Luxembourg
Email: joseluis.sanchezlopez@uni.lu

***Abstract*—This document [1] presents the Bachelor Semester Project made by Rafidison Santatra Rakotonsrasoa under the direction of his Project Academic Tutor Jose Luis Sanchez Lopez. BSP is a semester project that is to be done by any student in Bachelor in Computer Science (BiCS). For this semester (semester 6), our project is based on programming a person detection and person tracking program for a drone. The goal of this project is to develop a tool that can transform an affordable commercial drone with a monocular camera, that is easy to find in any store, into an autonomous aircraft that has the ability to track and follow a person. It is useful to know that some students supervised by our PAT did a similar project before, therefore, we want to start from their projects instead of starting from scratch. We investigate the existing face tracking and detection algorithms and analyse them. From the programming point of view, our PAT suggested that the programming part should be based on ROS, a Robotic Operating System framework that is very common in the field of robotic but also computer science especially when we want to manipulate robot, in our case, manipulating a drone. This report will explain and describe the methods we are using to achieve our goal.**

1. Introduction

Drones or Unmanned Aerial Vehicles (UAVs) have become a very popular research topic in the last few years. Many years ago, these remotely piloted or autonomous devices were developed for military purposes. UAV is a term that is commonly applied to military use cases. They are meant to fly and they can have a variety of purposes such as recreational purpose, photography, commercial, surveillance, defence or military. We will use a drone and work with it during this project. Drones can be automated or remotely piloted usually by smartphones, tablets, or joysticks or any other controllers which need somebody responsible who gives orders to the drone for making movements. The aim of this project is to make an autonomous drone, without the need for a handler that gives orders. We use a commercial drone, the DJI Tello drone, and we want it to detect a person and should be able to track and follow this person in real-time only based on the drone's camera.

In this paper, we can differentiate two different parts which are the scientific and the technical parts. The scientific part enclose all the scientific method and scientific discussions around our topic and the technical part, mainly, explain our code where we go deep in the implementation. Robot Operating System as known as ROS is used for the implementation part. ROS is the framework that we are using to implement our code. We will see later in this paper why we have decided to use it and what are the advantages of using it.

In the beginning, our work is based on some already existing face tracking and detection algorithms for a programmable drone. As a starting point, we got the projects of students that had similar projects. We analysed those projects and the goal was to investigate them and take the best of each project and codes and use them. However, apart from the investigation of those previous projects, more precisely the algorithms used to detect and track human faces using a drone with an on-board camera, our goal consisted of improving those existing person detection and tracking algorithms to overcome the existing limitations and to perform the estimation of the persons position and movement.

2. Project description

2.1. Domains

2.1.1. Scientific. The scientific domains of our project are described in this section. The domains we are interested in are computer vision related to person detection and tracking, and automation of drones. Computer vision is an approach to solve many tasks such as the detection of persons, faces, or various features, etc. Our main goal is to make an autonomous drone. An automated drone performs self-diagnosis that is capable of performing the requested flight, then takes off and navigates all without human input,

more specifically, to have a vision based person tracking and following drone. Always around this domain, we will also discuss the previous works and studies of students.

2.1.2. Technical. The technical domain of our project is described in this section. The domains we are interested in are the Robotic Operation System (ROS), the methods and algorithms we used to detect and track persons as well as controlling the drones. Their implementations are mainly based on ROS and python including the needed libraries. More details about the implementation are presented in the technical part of this paper.

2.2. Targeted Deliverables

2.2.1. Scientific deliverables. The scientific deliverables will answer the research question "What are person detection and tracking?". First, it is useful to understand what is a drone and what a drone can do, some features of the drone we are using should be presented. At beginning of this project, the previous works related to this project of four students were given to us to be analysed. Those projects are also given to start this project from a point at which something has been done yet and therefore to avoid doing everything from scratch.

2.2.2. Technical deliverables. The technical deliverables should be composed of the methods and the explanation of the implementation itself. We should present how the program has been implemented to accomplish its goal which is to detect and track a person.

3. Pre-requisites

In the sections below, we see the main scientific and technical knowledge that is required to be known before starting the project.

3.1. Scientific pre-requisites

For the scientific part, some knowledge about object detection and tracking can be a good asset since we will talk about it in the scientific part but those knowledge are not obligatory.

3.2. Technical pre-requisites

For the technical part, some knowledge in programming especially with the programming language *Python*. Therefore, it is required to be familiar with Python since we will mainly use this programming language for this project. We will use some *bash script*, just the basic scripts, so being familiar with it is also an advantage. We use ROS in our implementation, therefore, some background knowledge about it might help to understand easily our code.

4. Scientific deliverable 1

4.1. Requirements

The deliverables should present a scientific presentation of person detection and tracking, a brief analysis of the existing person detection and tracking. The main goal of the project is, given a commercial drone DJI Tello, to provide it with the capability of making an exact tracking of a person who is going to be in movement, and be able to follow it autonomously. To make the tracking and to give the drone the ability to follow the target we need computation work. For that, we connect a laptop with the drone and use this laptop to do computation work and send the information to the drone.

4.2. Design

Person detection

The first step that we will explain is the tracking part. We will see how we can select a region of interest. This part provides the tracker with a target to be tracked. We are working with a sequence of images or frames provided by the video, captured by the drone during the flight. Each frame is then processed and we want to detect a target in which we want to track afterwards. But for the moment, person detection is the goal here. Person detection is a process in our program which detects the person's face or particular key point (s) in the body of a person. In this project, we are going to use two different person detection algorithms from the computer vision field. The first algorithm is called *Pose Estimation* in which we detect the key points in a person's body/face. And the second algorithm is face detection using the Haar Cascade classifier.

Person tracking

The results from the detection process give information about the target we want to reach. The tracking process aims to handle the drone and gives the drone the instructions or commands to move accordingly. *PID Controller* or Proportional Integral Derivative controller will be used to track the target by generating steering commands. The PID Controller system shows a good result in the previous student projects that we have investigated. PID controller aims to make the drone stable, fast, precise and to limit oscillations. The PID comes into action when the target is detected by the drone, then it will calculate the speeds that this drone needs to have to recenter the target on the screen (in our case the target is a face). In their project,

their system was able to navigate in unknown environments without requiring artificial markers or external sensors, the system was entirely camera-based navigation. Therefore, we also decided to use the same system as them and use the PID controller.

4.3. Production

Keypoints detection using Pose Estimation algorithm

The pose estimation technique is made to estimate how a person is physically positioned such as raising the left hand, raising the right hand, standing or sitting. Instead of detecting and estimating the pose of the person, we are only interested in detecting the key points which are relevant to us. Here, we want to detect only the nose, the neck or the mid-hip of the person. This algorithm makes it possible to see the key points over the image. The figure 1 shows the *Keypoints* or *body points* and the joints that represent a human body.

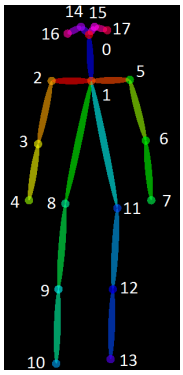


Figure 1. COCO keypoint format for human pose skeletons.

In our program, the index 0 of overall the body parts denote the *nose*, 1 for the *neck* and 8 for the *midhip*. We will use an already existing library for human pose estimation called *OpenPose* of tf-pose-estimation which stands for TensorFlow pose estimation. The modified version of the OpenPose project can be found here [2].

Face Detection using Haar Cascades

Object Detection using Haar feature-based cascade classifiers is an effective method proposed by Paul Viola and Michael Jones in the 2001 paper, "Rapid Object Detection using a Boosted Cascade of Simple Features". It is a machine learning based approach in which a cascade function is trained from a lot of positive images (images of faces) and negative images (images without faces). We extract the best features from those images using the Haar features shown in figure 2.

The best features are the ones that classify the best face and the non-face images. In the beginning, we have more than 160000 features and we need to select the best (about 6000) from them. For this, *Adaboost* classifier is used.

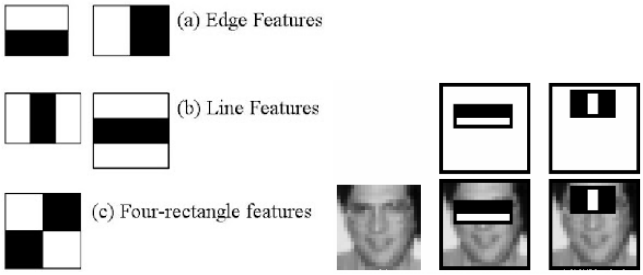


Figure 2. Haar features.

Adaboost classifier uses the boosting algorithm. Boosting refers to the process of sequentially training some homogeneous base models to build a model. Here, a base model depends on the previous ones. Predictions for boosting algorithms usually assign weights to determine the importance of each learner's input. The boosting process is, first, to train one base model, evaluate where the model or learner misclassified, then train another learner focusing on the areas where the previous learner misclassified and repeat this process until a stopping criteria.

Flight PID controller

PID (proportional-integral-derivative) controller is a control loop feedback mechanism commonly used in industrial control systems. In the real world, PID controllers are used from autonomous ship steering to drone autopilots. PID controller uses mathematical control theory to minimize the error rate over time by applying an adjustment or control variable. The idea is to design a stable and accurate controller for a drone. We want to use the PID control method to obtain stability in flying the quad-rotor drone.

Mathematically, the PID is the sum of the corrective actions, the Proportional Action *Kp*, Integral Action *Ki* and Derivative Action *Kd*. The equation is as follows:

$$PID = Kp*Error + Ki*\int_0^{\Delta t} Error + Kd*(dError/d\Delta t)$$

- *Kp* is the Proportional effect. The flight performance of the UAV can be summarised in two cases according to its Kp value. If Kp value is between 0 and 1, Kp is an attenuator, in this case, the stability of the drone is improved and, its oscillations are reduced, but the speed and precision are degraded. If Kp value is greater than 1, Kp is an amplifier, in this case, the speed and precision is improved, but the stability is decreased and its oscillations are increased.

- *Kd* is the Derivative effect. The more the Kd value increases, the more the movement execution time of the drone increases and, its oscillations are reduced.

- *Ki* is the Integral effect. The higher the Ki value is, the shorter the movement execution time of the drone, but there are more oscillations, and the stability of the drone is decreased.

- Δt is the delta time, the time that has passed between two following frames or the time differences between two frames.

- *Error* is the difference between the actual center of the face and the centre of the screen.

Drone flight controls

Most of the quad-rotor flying robots change their direction by manipulation of the individual rotor's speed and does not require cyclic and collective pitch control; the mechanical design is hence simpler and consequently reduces the production cost of the flying robot. We can distinguish 4 different types of flight controls that a drone can make: *yaw* (turning clockwise or anticlockwise), *roll* (left-right), *pitch* (forward-backwards) and *throttle* (up-down) controls.

To stay level in flight without moving sideways (hover), the thrust force from all four rotors should be exactly equal to the aircraft weight. To control the aircraft, the speed of each motor is changed. To make the drone go up or down, the power in each motor should be equally increased or decreased correspondingly. To go forward, backward, left, right or to rotate, the speed of the rotors is adjusted accordingly. The figure 3 shows us the different movements quad-copter drone can make. The blue color represents the normal speed of the rotor and the orange color represents high speed.

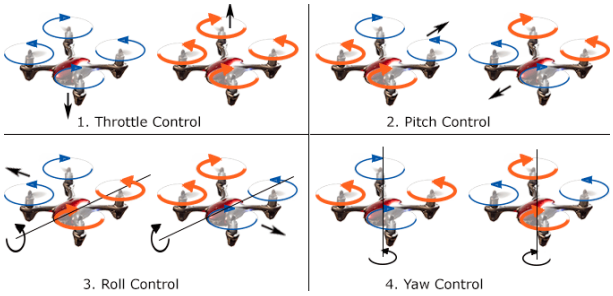


Figure 3. Quad-copter movements

4.4. Assessment

In the end, when we launch the drone and let it track a face, the simulation shows us that the PID controllers are able to robustly stabilize the quadrotor helicopter and move it to the desired position with the desired yaw, pitch, and throttle angles while keeping the roll angle zero. The same speed of response can be also seen in the yaw, pitch and roll angles control. The controller design is easy and, with a fast response time, can get the movements speed to its desired value. Through using the proposed PID controller method strategy, the good performance can be shown from the speed of response of the quadrotor.

5. Technical deliverable 1

5.1. Requirements

For the technical part of the project, the technical deliverables are composed of two main areas of interest. The first area is object/person detection and object/person tracking related to the computer vision field, making it with ROS. The second area is about the program that handles the drone according to the commands ordered by the person tracking process but also, we want to make it to be handle by the keyboard of a computer. The whole program is designed according to the ROS architecture by creating nodes and topics so that each process or node can be executed independently.

ROS or *Robot Operating System* is a middleware framework we can use to develop any robotics application such as robotic arm, mobile robot, drone and so on. It contains everything we need to build powerful robotics applications like framework, communication between processes, tools, plug and play plugins, active and growing community. The next generation of ROS called ROS2 or Robot Operating System 2 is more stable, with many new packages and functionalities released each month. In our project, the core concepts related to ROS are packages, nodes, topics, and messages, parameters, etc. And how to use them in your code.

5.2. Design

Initial system set up

Before starting to develop our system, we need to install and configure a set of tools. It may seem an obvious thing, but it took some time in our schedule. In our laptop, we have the operating system *Ubuntu 20.04* installed. We need to install ROS and download some packages such as OpenPose the pose estimation implementation package [2] and *tellopy* the python package which controls the DJI Tello drone [3] (the major portion of the source code was ported from the driver of GOBOT project [4])

We had chosen to install the *ROS Noetic* which is compatible with the version of ubuntu we have. It was necessary to learn the tools that we used to develop our program in case of not having the background of them. That is the reason why in our schedule we spent some time getting the knowledge in ROS, OpenPose and tellopy.

In addition, we need to install a series of libraries in python. For that, we had installed all the libraries we need in a virtual environment in anaconda, therefore we need to write the following code at the beginning of each python file we have:

```
1 #!/home/santatra/anaconda3/envs/Tf2_Py37/bin/
      python
```

Building the base system with ROS

We want to build the base system on which we want to run all the processes to make possible the tracking task. ROS allow us to run different processes of our system independently. Using *rqt graph* we can visualize the ROS graph of our application (see figure 6). We see all the running nodes, as well as the communication between them. The nodes and topics are displayed inside their namespace. The ROS runtime graph is a peer-to-peer network of processes. All the processes are run on our laptop but they can be distributed across different machines.

How ROS works?

A *node* is a process that can perform computation. Many nodes can communicate with one another using topics. Therefore, it is usually expected that we have many nodes. For example, in our case, one node controls the drone (named *driver*), one node performs the keyboard listener (named *keyboard*), one node performs the person detector (named *detector*), one node performs the person tracking computation (named *tracker*) and one node publishes the real-time video streams of the drone (named *tellocam_sub*).

Creating a catkin workspace: 'bsp6_ws'

A workspace is a directory containing ROS packages. This workspace will contain our system. *Catkin* is the official build system of ROS and the successor to the original ROS build system, *rosbuild*. Catkin generates 'targets' from raw source code that can be used by an end-user. These targets may be in the form of libraries, executable programs, generated scripts, and exported interfaces. Catkin is included by default when ROS is installed. Catkin can also be installed from source or prebuilt packages.

Before using ROS, it's necessary to source the ROS installation workspace in the terminal you plan to work in (see the following line of code). This makes ROS's packages available for you to use in that terminal.

```
source /opt/ros/noetic/setup.bash
```

Now we create the workspace folder by running the following bash scripts. *bsp6_ws* is the name of our workspace.

```
mkdir -p ~/bsp6_ws/src
cd ~/bsp6_ws/
```

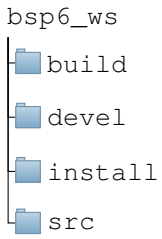
The next line of code will build any packages in the source space (/bsp6_ws/src) to the build space (/bsp6_ws/build). Any source files, python libraries, scripts or any other static files will remain in the source space.

```
catkin_make
```

To make sure your workspace is properly overlayed by the setup script, make sure ROS_PACKAGE_PATH environment variable includes the directory you're in by run the following code.

```
echo $ROS_PACKAGE_PATH
Results: /home/santatra/bsp6_ws/src:/opt/ros/
    noetic/share
```

As a result, we should get a folder tree.



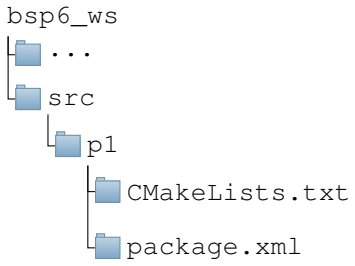
Creating a catkin package: 'p1'

A catkin package contains a catkin compliant *package.xml* file and a *CMakeLists.txt* file. The *package.xml* file which must be included with any catkin-compliant package's root folder and provides meta-information about the package such as the package name, version numbers, authors, maintainers, and dependencies on other catkin packages. The *CMakeLists.txt* file which is the input to the CMake building system for building software packages.

We use the catkin_create_pkg script to create a package called 'p1' which depends on std_msgs, rospy, image_transport, cv_bridge, sensor_msgs, std_msgs, actionlib, actionlib_msgs. Afterwards we build the package using the script catkin_make.

```
catkin_create_pkg p1 rospy image_transport
    cv_bridge sensor_msgs std_msgs actionlib
    actionlib_msgs
source /opt/ros/noetic/setup.bash
cd ~/bsp6_ws/
catkin_make
```

We should have the following folder tree.



Creating nodes:

'driver', 'keyboard', 'detector', 'tracker', 'tellocam_sub'

A ROS node can be a publisher or/and a subscriber node. A *publisher* node broadcasts messages continuously and puts the messages of some standard Message Type to a particular Topic. A *subscriber* node on the other hand subscribes to the Topic so that it receives the messages whenever any message is published to the Topic.

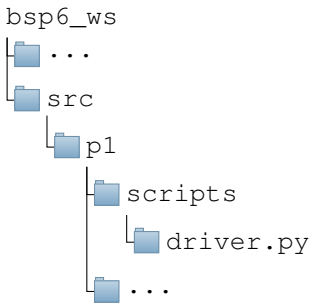
We need to create the folder 'scripts' in which will contain all our code files of the nodes.

```
roscd p1
mkdir scripts
cd scripts
```

After we implement the nodes, each node is implemented in a way that one node is in one python file. We use the script chmod +x to make the code file executable and build a node using catkin_make.

```
# <node_file> is the name of the node file e.g. driver.py
chmod +x <node_file>
cd ~/catkin_ws
catkin_make
```

We should have the following folder tree.



Creating message file:

'command_msgs.msg', 'op_mmsgs.msg'

Nodes communicate between them by sending and receiving ROS messages. There are many predefined messages such as sensor_msgs.msg.Image and from std_msgs.msg.String but we need two other customized messages in our implementation which are command_msgs and op_msgs. command_msgs are the messages the *tracker* node send to the *driver* node and op_msgs are the messages the *detector* node send to the *tracker* node.

We create a folder 'msg' which will contain all the message files

```
roscd <package_name>
mkdir msg
```

Then, we create the following message files:

command_msgs.msg

```
float64 yaw_vel
float64 roll_vel
float64 pitch_vel
float64 throttle_vel
```

op_msgs.msg

```
int32 nb_people
float64[] target_Nose
float64[] target_Neck
float64[] target_MidHip
float64[] face_center
float64 face_area
```

In package.xml, we add those lines:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

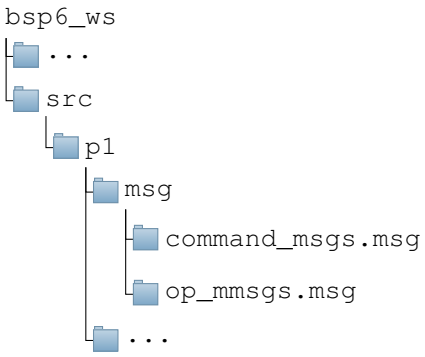
In CMakeLists.txt, add the following line in find_package:

```
message_generation
```

Still, in CMakeLists.txt, add the following lines in add_message_files:

```
command_msgs.msg
op_mmsgs.msg
```

We should have the following folder tree.



5.3. Production

The objective of this project is person detection and tracking using the onboard camera of a drone. To manage this project, we used the DJI Tello drone because it is easy to use and it will facilitate the communication between our program and the drone. It's partly due to the *tellopy* library that Tello give us, and this library is compatible with Python. Therefore, the code will be written in Python.

```
import tellopy
```

We need to import rospy when we are writing a ROS node. Therefore, we have to import it in each node file.

```
import rospy
```

To initialize a node we used the rospy.init_node method, it is very important as it tells rospy the name of the node. Until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. The name of a node must be a base name, i.e. it cannot contain any slashes "/".

```
rospy.init_node('name_of_the_node', anonymous=True)
```

Implementing the *driver* node

The *driver* node is a publisher and subscriber node. In this node, the main idea is to publish the image from the drone's camera to a potential node, to get the commands from the *keyboard* node and the commands from the *tracker* node. For this, we need a publisher to publish those image messages and two subscribers to get commands from other nodes.

We initialize the drone and we use *tellopy* library to control it.

```
self.drone = tellopy.Tello()
self.drone.connect()
self.drone.start_video()
```

The node publishes in real-time the image frames from the video captured by the drone's camera. Here the message type is Image, imported from sensor_msgs.msg. The sensor.msg import is so that we can reuse the sensor_msgs.msg/Image message type for publishing.

```
from sensor_msgs.msg import Image
```

The following code allows us to initialize the *driver* node

```
rospy.init_node('driver', anonymous=True)
```

Images are published to the topic, which makes *driver* a publisher. The following python code allows us to initialize the publisher for the image publishing. The node is publishing to the 'video_frames' topic using the message type Image. The queue_size argument limits the amount of queued messages if any subscriber is not receiving them fast enough.

```
self.pub = rospy.Publisher('video_frames', Image,
    queue_size=10)
```

To get the video stream captured by the drone, we use *av* package in python. This package is for direct and precise access to media via containers, streams, packets, codecs, and frames. It exposes a few transformations of that data and helps to get data to/from other packages. We use *OpenCV (cv2)* to process the image frames, hence, we convert the media to cv2.Image. We use *CvBridge* to convert ROS Image messages to OpenCV images and the other way around.

```
import av
import cv2
from cv_bridge import CvBridge

self.container = av.open(self.drone.
    get_video_stream())

br = CvBridge()
for frame in self.container.decode(video=0):

    # Convert frame to cv2 image
    frame = cv2.cvtColor(np.array(frame.to_image()
    ,dtype=np.uint8), cv2.COLOR_RGB2BGR)
    frame = cv2.resize(frame, (640, 480))
    self.pub.publish(br.cv2_to_imgmsg(frame))
```

The *driver* node get as a message the key names of the keyboard of the computer from the *keyboard* node. The node *driver* subscribes to the 'keypress' and 'keyrelease' topics to get every time we are pressing and releasing a key of the keyboard. Those topics are of type std_msgs.msgs.String. The following python code allows us to initialize the publishers.

```
rospy.Subscriber('keypress', String, self.
    keypress_callback)
rospy.Subscriber('keyrelease', String, self.
    keyrelease_callback)
```

When new messages are received, a callback is invoked with the message as the first argument. It means every time a key is pressed the method keypress_callback() is invoked, the same for keyrelease_callback(), it is invoked when a key is released. The following code shows the keypress_callback() and keyrelease_callback().

```
    def keypress_callback(self, key):
        if key == String("t"):
            print('{0} pressed'.format(key))
            self.drone.takeoff()
        elif key == String("l"):
            self.drone.land()
        elif key == String("o"):
            self.stop = True
        elif key == String("z"):
            self.set_speed("pitch", self.def_speed
    ["pitch"])
        elif key == String("s"):
            self.set_speed("pitch", -self.
    def_speed["pitch"])
        elif key == String("q"):
            self.set_speed("roll", -self.def_speed
    ["roll"])
        elif key == String("d"):
            self.set_speed("roll", self.def_speed[
    "roll"])
        elif key == String("a"):
            self.set_speed("yaw", -self.def_speed[
    "yaw"])
        elif key == String("e"):
            self.set_speed("yaw", self.def_speed["
    yaw"])
        elif key == String("p"):
            self.set_speed("throttle", self.
    def_speed["throttle"])
        elif key == String("m"):
            self.set_speed("throttle", -self.
    def_speed["throttle"])

    def keyrelease_callback(self, key):
        if key == String("z"):
            self.set_speed("pitch", 0)
        elif key == String("s"):
            self.set_speed("pitch", 0)
        elif key == String("q"):
            self.set_speed("roll", 0)
        elif key == String("d"):
            self.set_speed("roll", 0)
        elif key == String("a"):
            self.set_speed("yaw", 0)
        elif key == String("e"):
            self.set_speed("yaw", 0)
        elif key == String("p"):
            self.set_speed("throttle", 0)
        elif key == String("m"):
            self.set_speed("throttle", 0)
```

We set the keyboard commands to control the drone are set in the following way:

- 't': to take off
- 'l': to land
- 'z': to go forward
- 's': to go backward
- 'q': to go left
- 'd': to go right
- 'a': to rotate to the left

- 'e': to rotate to the right
- 'p': to go up
- 'm': to go down

The function set_speed set the speed of the drone according a certain axis. The axis can be *yaw*, *roll*, *pitch* or *throttle*.

```
    def set_speed(self, axis, speed):
        self.axis_speed[axis] = speed
```

The following code initializes the variables we need to control the drone. def_speed is the default value of the speed for each movement. axis_speed takes the values of the speed and is set to zero to initialize the speed for yaw, pitch, roll and throttle controls. prev_axis_speed is a variable that takes the previous speed values. The variable axis_command sends the speed values to the drone.

```
    self.def_speed =  { "yaw":50, "roll":35, "
    pitch":35, "throttle":80}

    self.axis_speed = { "yaw":0, "roll":0, "pitch"
    :0, "throttle":0}

    self.prev_axis_speed = self.axis_speed.copy()

    self.axis_command = {
        "yaw": self.drone.clockwise,
        "roll": self.drone.right,
        "pitch": self.drone.forward,
        "throttle": self.drone.up
    }
```

The object 'rate' with the help of its method sleep(), offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second as long as our processing time does not exceed 1/10th of a second. The while loop is a standard in ROS where we check if the node should still exist using the function rospy.is_shutdown(). The loop calls rate.sleep() which sleeps just long enough to maintain the desired rate through the loop. Every time, we get the keyboard or/and tracker commands, send those commands to the drone and publish images from the drone's camera.

```
rate = rospy.Rate(10)
br = CvBridge()
while not rospy.is_shutdown():

    for frame in self.container.decode(video=0):

        """GET KEYBOARD OR TRACKER COMMANDS"""
        for axis, command in self.axis_command.
    items():
            if self.axis_speed[axis]is not None
    and self.axis_speed[axis] != self.
    prev_axis_speed[axis]:
                command(self.axis_speed[axis])
                self.prev_axis_speed[axis] = self.
    axis_speed[axis]
            else:
                self.axis_speed[axis] = self.
    prev_axis_speed[axis]

        """PUBLISHING IMAGES"""
        frame = cv2.cvtColor(np.array(frame.
    to_image(),dtype=np.uint8), cv2.COLOR_RGB2BGR)
        frame = cv2.resize(frame, (640, 480))

        self.pub.publish(br.cv2_to_imgmsg(frame))

    rate.sleep()
```

Implementing the *keyboard* node

The *keyboard* node publishes the keys to the *driver* node every time we press on or release a keyboard key.

We initialize the node and the two publishers.

```
rospy.init_node('keyboard', anonymous=True)
self.pub_onpress = rospy.Publisher('keypress',
    String, queue_size=10)
self.pub_onrelease = rospy.Publisher('keyrelease',
     String, queue_size=10)
```

We use the *pynput* library to monitor the keyboard of our laptop. A *pynput.keyboard* listener checks if a keystroke is performed. The function rospy.spin() simply keeps python from exiting until this node is stopped.

```
from pynput import keyboard

def on_press(self, key):
    """ Handler for keyboard listener """
    print('{0} pressed'.format(key))
    self.pub_onpress.publish(str(key).strip('\''))

def on_release(self, key):
    """ Reset on key up from keyboard listener """
    print('{0} released'.format(key))
    self.pub_onrelease.publish(str(key).strip('\''
    ))
    if key == keyboard.Key.esc:
        # Stop listener
        return False

"""KEYBOARD LISTENER"""
with keyboard.Listener(
        on_press=self.on_press,
        on_release=self.on_release) as listener:
    listener.join()

rospy.spin()
```

Implementing the *detector* node

Basically, the *detector* node gets the image messages published by the *driver* node, detects the nose keypoint using the pose estimation, find the face using Haar cascades classifier and publish the coordinates of the nose and the information of the face.

We initialize the node, the subscriber which subscribes to the 'video_frames' topic in which the *driver* node publishes the image messages and the two publishers.

```
rospy.init_node('detector', anonymous=True)
rospy.Subscriber('video_frames', Image, self.
    callback)
self.pub_msgs = rospy.Publisher('op_msgs', op_msgs
    , queue_size=10)
self.pub_image = rospy.Publisher('op_frames',
    Image, queue_size=10)
def callback(self, data):
```

```
    br = CvBridge()
    self.image = br.imgmsg_to_cv2(data)
```

A function findFace(self, img) takes an image as a parameter and return the center of the face and the face area. The following findFace function was taken from the previous project of Mr Elliot KOCH and was adapted into our implementation.

```
def findFace(self, img):

    faceCascade = cv2.CascadeClassifier(cv2.data.haarcascades + 'haarcascade_frontalface_default.xml')
    imgGray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    faces = faceCascade.detectMultiScale(imgGray,1.2,4)

    myFaceListCenter = []
    myFaceListArea = []

    for (x,y,w,h) in faces:
        cv2.rectangle(img,(x,y),(x+w,y+h),(0,0,255),2)
        cx = x + w//2
        cy = y + h//2
        area = w*h
        myFaceListArea.append(area)
        if area == max(myFaceListArea):
            cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
        myFaceListCenter.append([cx,cy])

    if len(myFaceListArea) != 0:
        index = myFaceListArea.index(max(myFaceListArea))
        return img, [myFaceListCenter[index],myFaceListArea[index]]
    else:
        return img, [[0, 0],0]
```

The figure 4 is the result of the function findFace() with the center and the area of of the bounding box.

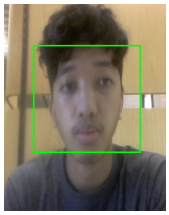


Figure 4. Bounding box on the face

The following implementation shows how we got the target nose of a person. We cloned the tf-pose-estimation by Ildoo Kim modified to work with Tensorflow 2.0+, the original repository can be found here [2]. In the pose estimation implementation, the neural network mobilnet_thin (Mobilenet V1) is used. The estimator *e* is created.

```
from tf_pose.estimator import TfPoseEstimator
from tf_pose.networks import get_graph_path, model_wh

model = "mobilenet_thin"
resize = "640x480"
tensorrt = "False"
resize_out_ratio = 4.0
w, h = model_wh(resize)
global e
if w > 0 and h > 0:
    e = TfPoseEstimator(get_graph_path(model), target_size=(w, h), trt_bool=str2bool(tensorrt))
else:
    e = TfPoseEstimator(get_graph_path(model), target_size=(640,480), trt_bool=str2bool(tensorrt))

humans = e.inference(self.image, resize_to_default=(w > 0 and h > 0), upsample_size=resize_out_ratio)
self.image = TfPoseEstimator.draw_humans(self.image, humans, imgcopy=False)
```

If we feed one image to the pose estimation function, we can obtain the human body skeleton drawn on the image (see figure 5).

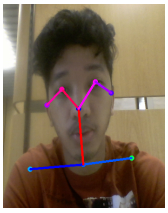


Figure 5. Body skeleton using Pose Estimation

We tested the combined Pose Estimation with Haar Cascades detection and a simple Haar Cascade detection, and we decided to go with the simple detection since the combined detection does not affect the detection process. Hence, instead of having the nose coordinates as a target, we use the center of the bounding box generated by Haar Cascades. The following implementation publishes the information of the face center and the face area from the *detector* node to the *tracker* node.

```
rate = rospy.Rate(10)
while not rospy.is_shutdown():
    if self.image is not None:

        img, face = self.findFace(self.image)
        self.msg.face_center = face[0]
        self.msg.face_area = face[1]
        print('face [{},{}]'.format(face[0], face[1]))
        """PUBLISHING FACE INFO"""
        self.pub_msgs.publish(self.msg)

    rate.sleep()
```

Implementing the *tracker* node

The *tracker* node get the op_msgs messages published by the *detector* node to the 'op_msgs' topic. Using a tracker function, we publish the command_msgs messages to the 'commands' topic which is subscribed by the *driver* node. With the following code, we initialize the node.

```
rospy.init_node('tracker', anonymous=True)
```

```
rospy.Subscriber('op_msgs', op_msgs, self.tracking_callback)
self.pub_cmds = rospy.Publisher('commands', command_msgs, queue_size=10)

def tracking_callback(self, msg):
    self.msg_face_center = msg.face_center
    self.msg_face_area = msg.face_area
```

Same as the *detector* node, we adapted the tracker function of Mr Elliot KOCH of the previous project on our implementation in the *tracker* node. To make the drone fly smoothly and not oscillate when it detects a face, a PID controller is used. A proportional Integral Derivative controller (PID controller) or PID corrector is a control system that improves the performance of a system whose principal purpose is to reach its target value as quickly as possible and to maintain it, regardless of external disturbances. Here the PID is going to improve the drone's flight performance by regulating its speed to have the following face in the centre of the stream. The tracker function is presented in the following code.

```
width_pid = [0.125,0.008,0.00025]
height_pid = [0.175,0.008,0.0001]

def width_Correction(previous_width_Error, width_Integral_Error, delta_time):
    global width_pid
    if previous_width_Error > 130 or previous_width_Error < -130 :
        # compute the width_Speed according to the previous_width_Error
        width_Speed = width_pid[0] * previous_width_Error + width_pid[1] * previous_width_Error/delta_time + width_pid[2] * (width_Integral_Error)
        width_Speed = int(np.clip(width_Speed,-80,80))
        return previous_width_Error, width_Speed
    else:
        return 0, 0

def height_Correction(previous_height_Error, height_Integral_Error, delta_time):
    global height_pid
    if previous_height_Error > 100 or previous_height_Error < -100  :
        # compute the height_Speed according to the previous_height_Error
        height_Speed = height_pid[0] * previous_height_Error + height_pid[1] * previous_height_Error/delta_time  + height_pid[2] * (height_Integral_Error)
        height_Speed = - int(np.clip(height_Speed,-80,80))
        return previous_height_Error, height_Speed
    else:
        return 0, 0

def track(face_center, face_area, w, h, width_pid, height_pid, for_back_pid, previous_width_Error, previous_height_Error, previous_for_back_Error, delta_time):
    global width_Integral_Error, height_Integral_Error, for_back_Integral_Error
    if face_center[0] != 0:
        width_Error = face_center[0] - w//2
        height_Error = face_center[1] - h//2
        for_back_Error = -(np.sqrt(face_area) - 90)

        # PID CONTROLLER
        width_Integral_Error += width_Error * delta_time
        height_Integral_Error += height_Error * delta_time
        for_back_Integral_Error += for_back_Error * delta_time

        width_Speed = width_pid[0] * width_Error + width_pid[1] * (width_Error - previous_width_Error)/delta_time + width_pid[2] * (width_Integral_Error)
        height_Speed = height_pid[0] * height_Error + height_pid[1] * (height_Error - previous_height_Error)/delta_time + height_pid[2] * (height_Integral_Error)
        for_back_Speed = for_back_pid[0] * for_back_Error + for_back_pid[1] * (for_back_Error - previous_for_back_Error)/delta_time + for_back_pid[2] * (for_back_Integral_Error)

        width_Speed = int(clip(width_Speed,-80,80))
        height_Speed = - int(clip(height_Speed,-80,80))
        for_back_Speed = int(clip(for_back_Speed,-40,40))

        yaw_velocity = width_Speed
        roll_velocity = 0
        pitch_velocity = for_back_Speed
        throttle_velocity = height_Speed

    else:
        height_Error, throttle_velocity = height_Correction(previous_height_Error, width_Integral_Error, delta_time)

        width_Error, yaw_velocity = width_Correction(previous_width_Error, height_Integral_Error, delta_time)

        for_back_Error = 0
        pitch_velocity = 0
        roll_velocity = 0

    vel = [yaw_velocity, roll_velocity, pitch_velocity, throttle_velocity]
    return width_Error, height_Error, for_back_Error, vel
```

Implementing the *tellocam_sub* node

The *tellocam_sub* node displays the images publish by the *driver* node to 'video_frames' topic. The following code is the implementation of the *tellocam_sub* node.

```
rospy.init_node('tellocam_sub_py', anonymous=True)
rospy.Subscriber('video_frames', Image, callback)
rospy.spin()

def callback(data):
  br = CvBridge()
  current_frame = br.imgmsg_to_cv2(data)
  cv2.imshow("camera", current_frame)
  cv2.waitKey(1)
```

5.4. Assessment

For the technical aspects of the project, we provided more amount of work and more programming work in ROS since we were not familiar with it before this project and we also needed to build the system for our program over ROS.

We built the system over ROS in addition to the person detection and tracking algorithm. Fortunately, thanks to the previous project, we had an already existing program for detection and tracking and we needed to adapt it with our system and the second detection algorithm, the Pose Estimation. However, during the test we did, despite Pose Estimation gives accurate results of the target, combining Pose Estimation and Haar Cascade classifier does not give necessarily the best results than a simple Haar Cascade classifier to compute both the target and the bounding box of a face. We think that this might due to the fact that estimating the pose of a person might not be the best solution to get a target since the whole pose estimation program is time-consuming comparing to the Haar Cascade classifier. In general, the technical requirements are satisfied, the drone is able to detect and track the potential target and move accordingly and, more importantly, the implementation follows the ROS architecture.

Acknowledgment

I would like to thank the BiCS course director Mr Nicolas GUELFI, my Project Academic Tutor Mr Jose Luis SANCHEZ LOPEZ, Mr Elliot KOCH and the BiCS management and education team, for their help, for their support, for their time spent helping me throughout the project and for the advice concerning the missions mentioned in this report and Mr.

6. Conclusion

The objective of this project was to create a program of person detection and tracking with an on-board camera of a drone. To complete this objective and realize the program we first had to understand the methods of it.

The scientific part of this project was based on the detection and the tracking part. Understand how a computer which have only pixel as data, can see a face and can track it using only images. For the tracking we have seen how the Pose Estimation and the Haar Cascades are used in our project and also how to control the drone.

The technical part of this project was based on the implementation. First, we made our system ready to use ROS and then afterwards we learned how ROS works. Next, we implemented the different nodes in our system.

This Bachelor Semester Project was, for me, an excellent opportunity to learn how to design and build a tracking person program for drone.

References

[1] Nicolas Guelfi. Bics bachelor semester project report template, 2017. University of Luxembourg, BiCS - Bachelor in Computer Science, https://github.com/nicolasguelfi/lu.uni.course.bics.global.

[2] Ildoo Kim. Tensorflow approach for the pose estimation. https://github.com/gsethi2409/tf-pose-estimation.

[3] Hanyazou. Dji tello drone controller python package. https://github.com/hanyazou/TelloPy.

[4] Gobot is a framework for robots, drones, and the Internet of Things (IoT), written in the Go programming language https://github.com/hanyazou/TelloPy.

7. Appendix

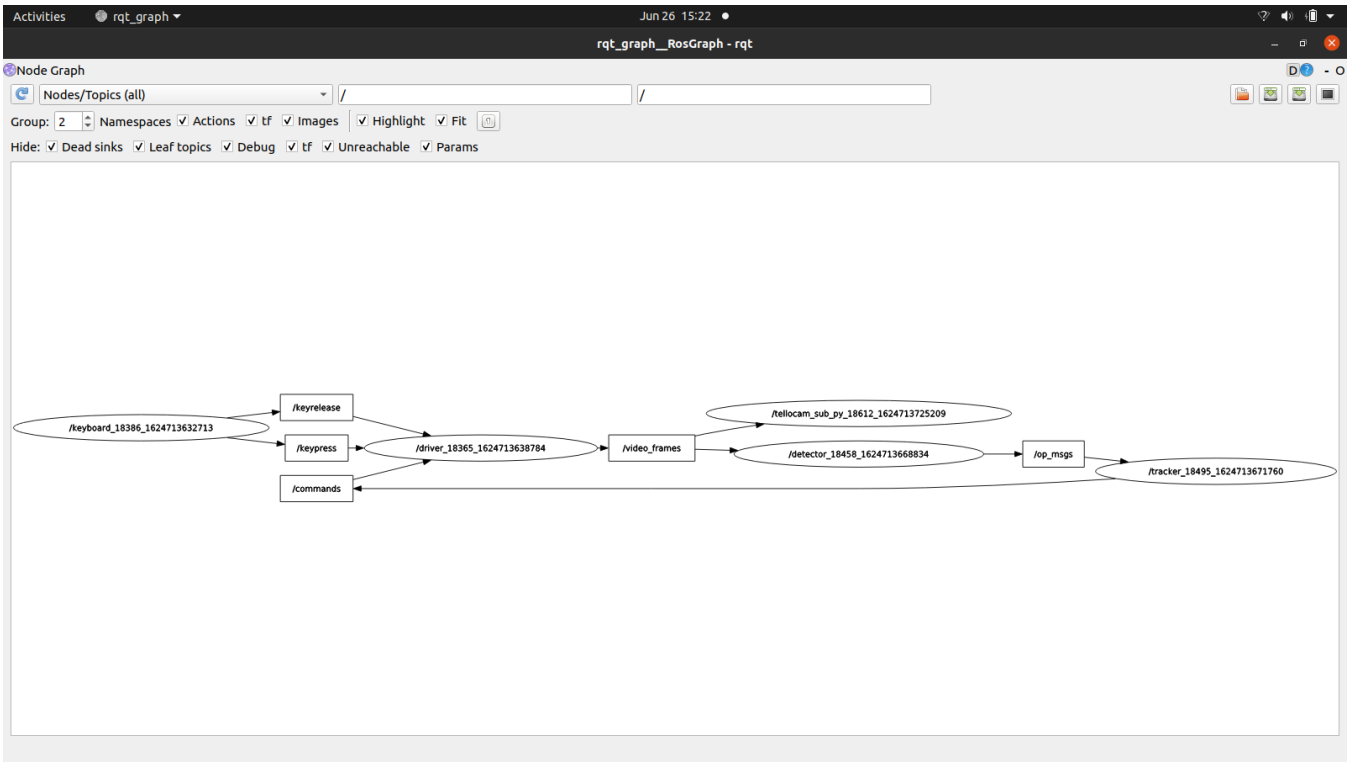


Figure 6. ROS graph of our application generated by rqt graph