Report: Pluto Drone Swarm Challange

https://github.com/Inter-IIT-Drona-Aviation/final_Python_wrapper

TASK 1: Control Drone using a custom made python wrapper

Our wrapper (in the link above) contains three modules:

- DroneConnector (handles telnet communication and writes MSP packets to the telnet channel)
- MSPHelper (defines MSP commands and their structure)
- Pluto (implements drone control using MSP commands)

except for the module, the wrapper contains a demo example.

The following functions are implemented in the wrapper:

- Connect to drone
- ARM
- Take off
- Throttle up & down
- Pitch forward & backward
- Roll right & left
- Yaw right & left
- Land
- Disarm
- Disconnect from drone

1. DroneConnector Module:

In this module, "__init__.py "maintains connection and sends MSPdatapackets to the drone using "telnetlib" library.

The following functions are implemented:

- Connect to the drone with the given "IP" and "Port"
- Make an MSPdatapacket and Send it to the drone
- Disconnect from drone

- Reconnect to drone

The following code is used to make a byte array, get the checksum, and create MSPdatapackets:-

```
# make a byte array as per the requirnment of multiwii protocol and telnet communication
def concatByteArrs(*arrs):
    res = b''
    for i in arrs:
      res += i

    return res

# perform bitwise xor to find the checksum for MSP commands
def getChecksum(*args):
  byteArr = concatByteArrs(*args)

  checksum = 0
  for byte in byteArr:
    checksum ^= byte
  # struct.pack return a object from the parameter in given format
  return struct.pack("<B", checksum)

# create and return data packet (byte array) based on multiwii serial protocol
# (header '$M', direction '>/<', msgLength, payloadType, payload, checksum)
def createDataPacket(direction, msgLength, payloadType, payloadFormat, payload):
  HEADER = struct.pack("<2c", b'$', b'M')
  DIRECTION = struct.pack("<c", direction.encode('utf-8'))
  MSG_LENGTH = struct.pack("<B", msgLength)
  PAYLOAD_TYPE = struct.pack("<c", payloadType)
  PAYLOAD = struct.pack(payloadFormat, *payload)
  CHECKSUM = getChecksum(MSG_LENGTH, PAYLOAD_TYPE, PAYLOAD)

return concatByteArrs(HEADER, DIRECTION, MSG_LENGTH, PAYLOAD_TYPE, PAYLOAD, CHECKSUM)
```

- concatByteArrs method concate array of byte strings into a single byte string
- checksum method gives the bitwise xor of MSG_LENGTH, PAYLOAD_TYPE, PAYLOAD
- createDataPacket method formats all the portions of the MSP message and returns it as byte string.

2. MSPHelper Module:

This module defines the MSP commands keeping in mind the command structure and required arguments.

The MSP command structure is as follows:

- Header (char), Direction (char), Msg length (hex), Type of packet (hex), Payload (hex), Checksum (hex)

  More details on the MSP commands at:
  https://docs.google.com/document/d/1c2tjbeAuTYk3JZrkazImayqjCKx9w3rND4RTN41ol6U/edit#heading=

  or

  https://create.dronaaviation.com/software/remote-programming/make-your-own

All 6 data are together in a byte array, and hex values are separated by a backward slash '\' before forming the byte string. for example, the MSP_SET_RAW_RC will look like this:

```
b'$M' # Header
b'<'  # Direction
b'\x10' # msg length (16 bytes) (hex of 16 is 10)
b'\xc8' # type of packet
# 2 byte for each decimal number, total 16 for an array of 8 numbers
# so \xdc\x05 has 1st byte xdc(11011100) and 2nd byte x05(00000101) which together is (0000010111011100)=1500
# given byte string is for the following array [1500, 1500, 1500, 1500, 900, 900, 900, 900]
b'\xdc\x05\xdc\x05\xdc\x05\xdc\x05\x84\x03\x84\x03\x84\x03\x84\x03'
b'\xd8' # checksum

# final byte string to be sent to over the communiaction channel
b'$M<\x10\xc8\xdc\x05\xdc\x05\xdc\x05\xdc\x05\x84\x03\x84\x03\x84\x03\x84\x03\xd8'
```

Here the command specific data is provided by MSPHelper Module and the byte array is made by concateByteArrs, getChecksum, and createDataPacket methods of DroneConnector Module.

3. Pluto Module:

This module impliments all the function for controlling the drone.

Implimented commands are:

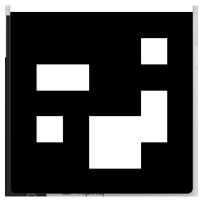
- Arm, Disarm, Restart: to arm, disarm and soft restart (disarm→reconnect→arm) the pluto.
- Throttle, Pitch, Roll, Yaw: all has a single parameter, the specific value for the command (900-2100).
- Hower: to hower in altitude hold mode.
- up, down, forward, backward, left, right, turnLeft, turnRight: all have one parameter, the amount of time in sec to keep running the command (for beginner users hence no option to control the intensity of the command).

💡 The use of these function in a code is shown in the demo.py file

TASK 2: Aruco pose estimation, PID implimentation

A. Generate the ArUco tag and place it on the drone.

The Aruco tag that we are using is :
The dimensions of tag is 5cm*5cm, pasted on the Pluto drone.



B. Pose estimation of drone:

I. INSTALLATION:

We are using Visual Studio Code as the IDE for running our codes.

Verify if Pip is installed: Go to the folder where python is present. Then open that location in the terminal and enter:

```
$ pip --version
```

It will return the respective version if present. If not,

To install PIP on your system, write the following commands on your Terminal(on Desktop):

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python get-pip.py
```

- OpenCV: OpenCV is a huge open-source library for computer vision, machine learning, and image processing. Open your terminal and enter:

```
$ pip uninstall python-opencv opencv-contrib-python opencv-python opencv-python-headless
$ pip install opencv-contrib-python==4.6.0.66
```

This will initiate the installation process. Wait for the installation to complete.

Now check if OpenCV is correctly installed:

```
$ python
>>>import cv2
>>>print(cv2.__version__)
```

- NumPy: It is a general-purpose array processing package that provides tools for handling n-dimensional arrays. This comes pre-installed with python 3.11.0.

```
$ pip install numpy
```

After installing all the dependencies, get the code from the GitHub link provided at the starting of the doc.

Create a folder, and write the following command in gitbash:

```
$ git init
$ git clone https://github.com/Inter-IIT-Drona-Aviation/aruco_pose_estimation
```

Open the files with VS Code.

II. CODE FOR POSE ESTIMATION:

Class=aruco_class.py

This code is a class named "**aruco**" that performs ArUco marker detection and pose estimation and gives the data in the video stream.

a**ruco_dict** is a dictionary that maps strings names to aruco dictionary constants defined in OpenCV library. These constants specify the type of aruco marker dictionary to use for detection. This must be changed based on the ArUco tag used.

Set the intrinsic camera matrix **self.intrinsic_camera** and distortion coefficients **self.distortion** to a predefined value. The intrinsic camera matrix represents the intrinsic parameters of the camera used for capturing the video stream. The distortion coefficients represent the radial and tangential distortion of the camera lens. Specific to the camera being used.

```
def pose_estimation(self, frame):
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
        cv2.aruco_dict = cv2.aruco.Dictionary_get(ARUCO_DICT[self.aruco_type])
        parameters = cv2.aruco.DetectorParameters_create()
corners, ids, rejected_img_points = cv2.aruco.detectMarkers(gray, cv2.aruco_dict, parameters=parameters )
```

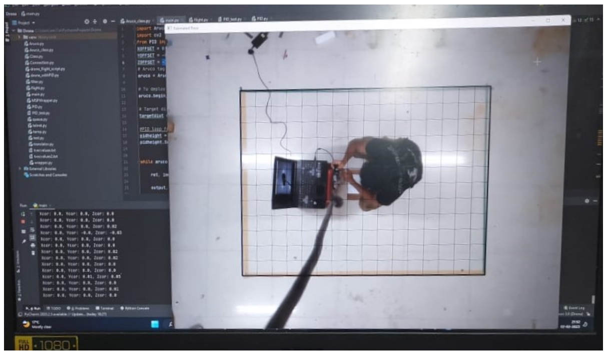
The **pose_estimation** function takes a frame as input, converts it to grayscale, detects ArUco markers in the frame using the selected dictionary and parameters, and estimates the pose of each marker if any markers are detected.

```
if len(corners) > 0:
            for i in range(0, len(ids)):
                rvec, tvec, markerPoints = cv2.aruco.estimatePoseSingleMarkers(corners[i], 0.02, self.intrinsic_c
amera,
                                                                              self.distortion)

                cv2.aruco.drawDetectedMarkers(frame, corners)
            return frame, rvec, tvec
        return frame, None, None
```

The function returns the input frame with the detected markers drawn on it, and the rotation and translation vectors for each marker.

Arena



aruco_poseestimation.py

Important aspects of the script are explained below:

```
def rotation_matrix_to_attitude_angles(R):
```

The above function is used to retrieve the yaw, pitch and roll angles from the rotation matrix generated by the Rodrigues function. Provision for gimbal lock condition is present in the form of the else statement. The integer value of the angle in Degrees in returned. Last line of the code is application dependent.

```
        transcoord[0] = round(trans[0][0][0] + XOFFSET, 2)
        transcoord[1] = round(trans[0][0][1] + YOFFSET, 2)
        transcoord[2] = round(ZOFFSET - trans[0][0][2], 2)

        #get the rotation matrix from rvec
        rmat = cv2.Rodrigues(rot)[0]

        #get the yaw, pitch, roll in order
        rotcoord = rotation_matrix_to_attitude_angles(rmat)
        # print(f"Xcor: {transcoord[0]}, Ycor: {transcoord[1]}, Zcor: {transcoord[2]}")
        print(f"Filtered: Xcor: {filters[0].filter(transcoord[0])}, Ycor: {filters[1].filter(transcoord[1])}, Zcor:
        # print(rotcoord)
        print(f"Filtered: Yaw: {filters[3].filter(rotcoord[0])}, Pitch: {filters[4].filter(rotcoord[1])}, Roll: {fil
```

The various operations on the coordinates are performed here.

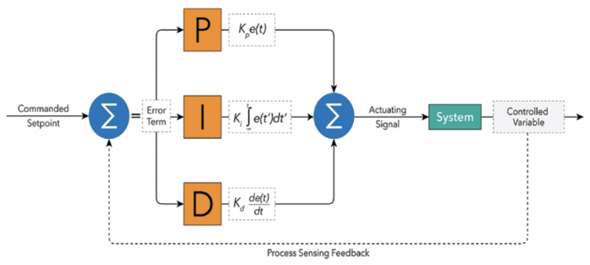
1. First the coordinates are shifted from the camera plane to the plane of our setup. The point right below the camera is chosen as [0, 0, 0]. This is implemented using the offsets introduced at the beginning of the script.
2. The rotation matrix is retrieved from the rvec values by the Rodrigues function which is then handed over to the function defined above, to generate yaw, pitch and roll.
3. A moving average filter is implemented to produce steady output.

```
    # Grid to locate the aruco in the our camera setup
    start_point = (0, 0)
    end_point = (1280, 960)
    start_point2 = (1040, 780)
    end_point2 = (240, 180)
    color = (0, 0, 0)
    thickness = 1
    output = cv2.rectangle(output, start_point, end_point, color, thickness)
    output = cv2.rectangle(output, start_point2, end_point2, color, thickness*3)
    for i in range(240, 1040, 50) :
        output = cv2.line(output, (i, 180), (i, 780), color, thickness)
    for i in range(180, 780, 50) :
        output = cv2.line(output, (1040, i), (240, i), color, thickness)
    cv2.circle(output, (640,480), 5, color, thickness)
    cv2.imshow('Estimated Pose', output)
```

OpenCV's drawing capability is used to generate a grid for our setup. Used for verification of measurements. Should be commented out or changed with respect to the setup being used.

C. Add PID to the script for controlling the drone

PID Class can be found in the directory named ***PID.py***



The standard PID loop takes the following inputs:

1. Setpoint:
   The setpoint is handed over as an attribute in the class.
2. PID coefficients
   The control coefficients are given during the initialization of the object.

Important Aspects of the PID class defined:

1. To run an iteration of the PID loop, the method ***.update(feedback)*** is called. The feedback value is handed over as an argument. This generates the output and stores it in the attribute ***.output*** The value is

then handed over to the hover function.

2. The sample time of the PID loop can be implemented for regular loops. By default it is set as 0. It can be set, using the method .***setSampleTime().***
3. A windup guard is also implemented to keep the integral bounded and prevent large overshoots.

D. Hover the drone in one position.

We have created function for hovering after testing.

“hover function” is keeping the drone at particular height for particular time.

```
    pidheight_z.update(zcor)
    drone.hover(0,1500+round(pidheight_z.output))
```

Hover Function accepts an input generated by the PID loop. The extra throttle is added to the steady state value to generate a deviation about the steady state value.

```
def hover(self,t,k=1500):
        self.__updatepose(1500,1500,k,1500,1500,t)
```