PROJECTS WITH PLUTO USING PYTHON

Version 1

+

python

Table of Contents

Introduction

Pluto is not just a drone, it is a development tool. It enables the user to develop numerous projects across different areas. With the Pluto Platform, you can create and innovate to bring your ideas into reality. From simply glowing LEDs, to understanding complicated scientific concepts, to developing fun games - everything is possible with the Pluto Python Platform. The sensor data and the remote-control data are all accessible with python, making it one of a kind.

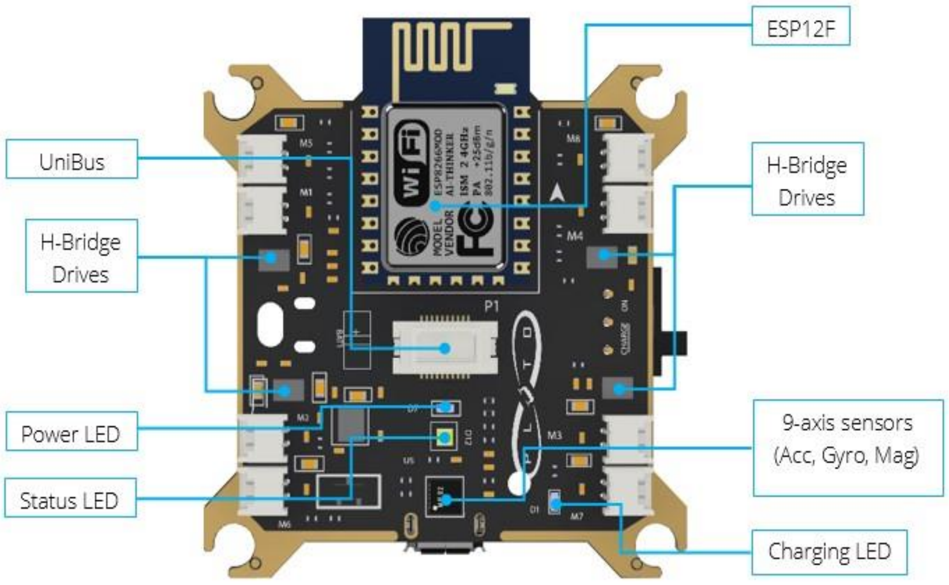
With the platform being open source, tinkering is made a lot easier with the predefined python functions. Additionally, for those who prefer Python, the Pluto Platform offers extensive support for Python programming. This makes it easier to integrate various libraries and tools, providing a versatile environment for developers.

Some key features of the Pluto Platform are:

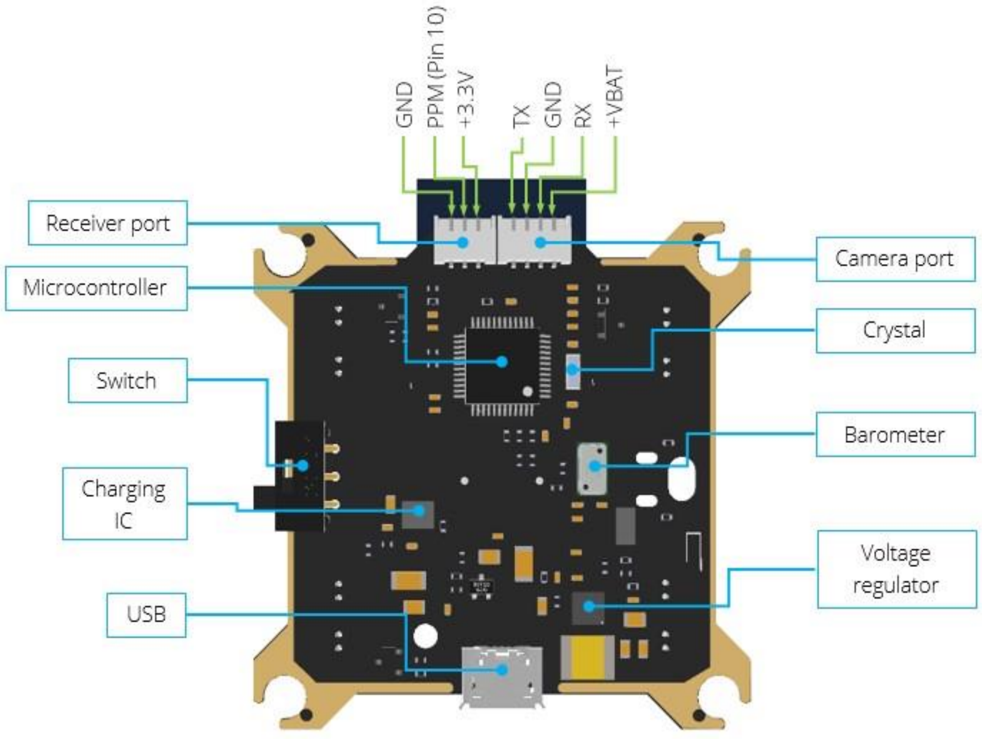
✓ Easy predefined functions in Python
✓ Real-time sensor data access
✓ Remote control data integration
✓ camera access and control
✓ opencv image processing

The Pluto Platform aims to motivate users to begin their journey into the universe of drones, making it accessible and enjoyable for both beginners and experienced programmers.

Board Details



Primus X (TOP)



Primus X (BOT)

Compatibility

The Projects with Pluto: Using Python contains projects that are built using python and executed using Pluto nano drone. The compatibility for both is given below.

Operating System:

- Windows
- Linux
- MacOS

Python Versions:

- Any version greater than 3.7.0

Drones:

- Pluto X nano drone
- Pluto 1.2 nano drone

Other:

- Rover
- Hover

Python Installation and Setup

For Windows Users:

1. **Download Python**: Visit the official Python website (python.org) and download the latest version of Python for Windows. Choose the executable installer for simplicity.

2. **Run the Installer**: Double-click the downloaded file to start the installation. Ensure to check the box that says "Add Python 3.x to PATH" to make Python accessible from the command line.

3. **Verify Installation**: Open Command Prompt and type python --version. If Python is successfully installed, you'll see the version number displayed.

4. **Note:** Avoid using the Microsoft store version its not very efficient for development.

For Linux Users:

1. **Check for Pre-installed Python**: Most Linux distributions come with Python pre-installed. Open a terminal and type python --version or python3 --version.

2. **Install Python**: If Python is not installed or you need a different version, use your distribution's package manager. For Ubuntu or Debian-based systems, use sudo apt-get update followed by sudo apt-get install python3.

3. **Verify Installation**: After installation, type python3 --version in the terminal to confirm the Python version.

For MacOS Users:

1. **Check for Pre-installed Python**: macOS comes with Python pre-installed. Open Terminal and type python --version or python3 --version to check the installed version.

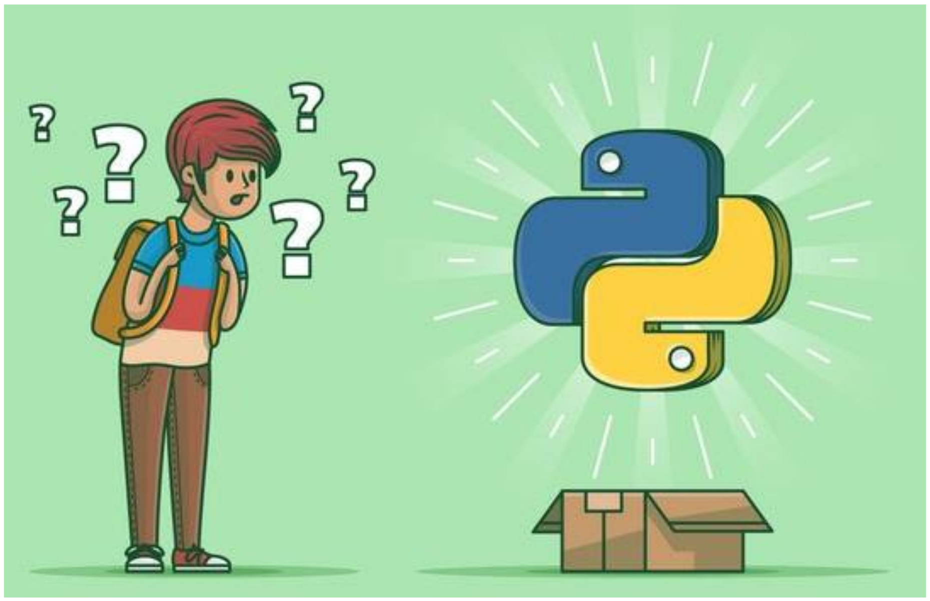
2. **Install Latest Python (optional)**: If you need the latest version, visit python.org to download the macOS installer. Follow the installation prompts.

3. **Use Homebrew (alternative)**: Install Homebrew (a package manager for macOS) by pasting /bin/bash-c"$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)" in Terminal. Then, install Python by typing brew install python.

Chromebooks: With newer Chromebooks that support Linux apps, you can install Python via the Linux terminal using standard Linux commands to install Python.

Why Use Python?

- Python's simplicity, versatility, and rich libraries make it a popular choice for drone programming.
- Its straightforward syntax facilitates quick learning and coding, with a wide array of pre-built tools available for tasks such as flight path analysis, sensor data processing, and system integration.
- Python's seamless compatibility with robust image processing libraries like OpenCV further enhances its suitability for drone programming.
- Cross-platform Support: Python's ability to run on multiple operating systems such as Windows, macOS, and Linux ensures that drone developers can build and test their software across different environments without needing significant changes to the codebase.
- Strong Community and Resources: Python benefits from a vast and active community of developers. This means programmers working on drone technology can easily find support, tutorials, and pre-existing projects. This extensive community also leads to a rich ecosystem of plugins and extensions that are particularly useful for handling various drone-related functionalities.
- Real-time Data Handling: Python’s efficient handling of real-time data is critical for drone operations which rely heavily on sensor inputs. Libraries like NumPy and Pandas simplify the task of managing and analyzing real-time data from drones, enabling more effective decision-making during flight.



Pluto Python Package - Plutocontrol

Plutocontrol is a Python library for controlling Pluto drones. This library provides various methods to interact with the drone, including connecting, controlling movements, and accessing sensor data.

plutocontrol allows you to interact with Pluto drones using MSP (Multiwii Serial Protocol) commands and perform various actions, such as changing its flight mode, setting motor speeds, or reading sensor data.

Installation:

pip install plutocontrol

Usage:
After installing the package, you can import and use the Pluto class in your Python scripts.

Example Usage:

```
from plutocontrol import Pluto

# Create an instance of the Pluto class
pluto = Pluto()

# Connect to the drone
pluto.connect()

# Arm the drone
pluto.arm()

# Disarm the drone
pluto.disarm()

# Disconnect from the drone
pluto.disconnect()
```

Class and Methods

Pluto Class

Connection
Commands to connect/ disconnect to the drone server.

```
#Connects from the drone server.
pluto.connect()

#Disconnects from the drone server.
pluto.disconnect()
```

Camera module
Sets the IP and port for the camera connection.

```
#should be intialized before pluto.connect().
pluto.cam()
```

Arm and Disarm Commands

```
#Arms the drone, setting it to a ready state.
pluto.arm()

#Disarms the drone, stopping all motors.
Pluto.disarm()
```

Pitch Commands

```
#Sets the drone to move forward.
pluto.forward()

#Sets the drone to move backward.
pluto.backward()
```

Class and Methods

Roll Commands

```
#Sets the drone to move left (roll).
pluto.left()

#Sets the drone to move right (roll).
pluto.right()
```

Yaw Commands

```
#Sets the drone to yaw right.
pluto.right_yaw()

#Sets the drone to yaw left.
pluto.left_yaw()
```

Throttle Commands

Increase/ Decrease the drone's height.

```
#Increases the drone's height.
pluto.increase_height()

#Decreases the drone's height.
pluto.decrease_height()
```

Takeoff and Land

```
#Arms the drone and prepares it for takeoff.
pluto.take_off()

#Commands the drone to land.
pluto.land()
```

Class and Methods

Developer Mode

Toggle Developer Mode

```
#Turns the Developer mode ON
pluto.DevOn()

#Turns the Developer mode OFF
pluto.DevOff()
```

Motor Commands

```
#motor_speed(motor_index, speed)
#Sets the speed of a specific motor
#(motor index from 0 to 3).

pluto.motor_speed(0, 1500)
```

Get MSP_ALTITUDE Values

```
#Returns the height of the drone from the sensors.
height = pluto.get_height()

#Returns the rate of change of altitude from the sensors.
vario = pluto.get_vario()
```

Get MSP_ALTITUDE Values

```
#Returns the roll value from the drone.
roll = pluto.get_roll()

#Returns the pitch value from the drone.
pitch = pluto.get_pitch()

#Returns the yaw value from the drone.
yaw = pluto.get_yaw()
```

Class and Methods

Get MSP_RAW_IMU Values

Accelerometer

Returns the accelerometer value for the x,y,z - axis.

```
#Returns the accelerometer value for the x-axis.
acc_x = pluto.get_acc_x()

#Returns the accelerometer value for the y-axis.
acc_y = pluto.get_acc_y()

#Returns the accelerometer value for the z-axis.
acc_z = pluto.get_acc_z()
```

Gyroscope

Returns the Gyroscope value for the x,y,z - axis.

```
#Returns the Gyroscope value for the x-axis.
gyro_x = pluto.get_gyro_x()

#Returns the Gyroscope value for the y-axis.
gyro_y = pluto.get_gyro_y()

#Returns the Gyroscope value for the z-axis.
gyro_z = pluto.get_gyro_z()
```

Magnetometer

Returns the Magntometer value for the x,y,z - axis.

```
#Returns the Magnetometer value for the x-axis.
mag_x = pluto.get_mag_x()

#Returns the Magnetometer value for the y-axis.
mag_y = pluto.get_mag_y()

#Returns the Magnetometer value for the z-axis.
mag_z = pluto.get_mag_z()
```

Class and Methods

Calibration Commands

```
#Calibrates the accelerometer.
pluto.calibrate_acceleration()

#Calibrates the magnetometer.
pluto.calibrate_magnetometer()
```

Get MSP_Analog Values

```
#Returns the battery value in volts from the drone.
battery = pluto.get_battery()

#Returns the battery percentage from the drone.
battery_percentage = pluto.get_battery_percentage()
```

Script based controls

1. **Throttle Commands – takeoff and land**

```
from plutocontrol import pluto
import time

# Initialize the drone
drone = pluto()

# Connect to the drone
print("Connecting to the drone")
drone.connect()
time.sleep(2)

# Arm the drone
drone.arm()
time.sleep(3)

# Take off
drone.take_off()
time.sleep(1)

for _ in range(50):
    drone.increase_height()
    time.sleep(0.05)

# Hover for a bit before landing
print("Hovering")
time.sleep(5)

# Land the drone
print("Landing the drone")
drone.land()
time.sleep(5)

drone.disconnect()
```

Script based controls

2. SensorLogger

```
import time
from plutocontrol import pluto
drone = pluto()

# Assuming you've instantiated your drone object as 'drone'
drone.connect()

drone.get_battery()

def continuously_get_sensor_values(interval_seconds=1):
    while True:
        mag_x = drone.get_mag_x()
        mag_y = drone.get_mag_y()
        mag_z = drone.get_mag_z()
        gyro_x = drone.get_gyro_x()
        gyro_y = drone.get_gyro_y()
        gyro_z = drone.get_gyro_z()
        acc_x = drone.get_acc_x()
        acc_y = drone.get_acc_y()
        acc_z = drone.get_acc_z()

        print(f"Magnetometer (X,Y,Z): ({mag_x}, {mag_y}, {mag_z})")
        print(f"Gyroscope (X,Y,Z): ({gyro_x}, {gyro_y}, {gyro_z})")
        print(f"Accelerometer (X,Y,Z): ({acc_x}, {acc_y}, {acc_z})")

        time.sleep(interval_seconds)

# Call the function to start retrieving sensor values continuously
continuously_get_sensor_values()
```

External controllers – Keyboard Control

Keyboard control

To set up keyboard controls for your Pluto drone, follow these general steps:

For windows:

keyboard_control.py

```
from plutocontrol import pluto

my_pluto = pluto()

def identify_key(key):
    if key == 70:
        if(my_pluto.rcAUX4 == 1500):
            my_pluto.disarm()
        else:
            my_pluto.arm()
    elif key == 10:
        my_pluto.forward()
    elif key == 30:
        my_pluto.left()
    elif key == 40:
        my_pluto.right()
    elif key == 80:
        my_pluto.reset()
    elif key == 50:
        my_pluto.increase_height()
    elif key == 60:
        my_pluto.decrease_height()
    elif key == 110:
        my_pluto.backward()
    elif key == 130:
        my_pluto.take_off()
    elif key == 140:
        my_pluto.land()
    elif key == 150:
        my_pluto.left_yaw()
    elif key == 160:
        my_pluto.right_yaw()
```

External controllers – Keyboard Control

Keyboard.py

```
import os
import sys
import keyboard_control
import keyboard as kb

##########################################################
# Key Controls

# c : Connect
# spacebar : arm or disarm
# w : increase height
# s : decrease height
# a : yaw left
# d : yaw right
# q : take off
# e : land

# n: To enter developer mode

# Up arrow : go forward
# Down arrow : go backward
# Left arrow : go left
# Right arrow : go right
# e: to quit
##########################################################

if os.name == 'nt':  # Windows
    def getKey():
        event = kb.read_event()
        key = event.name
        if event.event_type == kb.KEY_DOWN:
            if key == 'up':
                key = '[A'
            elif key == 'down':
                key = '[B'
            elif key == 'left':
                key = '[D'
            elif key == 'right':
```

External controllers – Keyboard Control

```
            key = '[C'
         elif key == 'space':
            key = ' '
# print(key)
         return key

keyboard_cmds={  #dictionary containing the key pressed and
value associated with it
                '[A': 10,
                '[D': 30,
                '[C': 40,
                'w':50,
                's':60,
                ' ': 70,
                'r':80,
                't':90,
                'p':100,
                '[B':110,
                'n':120,
                'q':130,
                'e':140,
                'a':150,
                'd':160,
                '+' : 15,
                '1' : 25,
                '2' : 30,
                '3' : 35,
                '4' : 45}

while True:
   key = getKey()
   if key == 'e':
      print("stopping")
      break
   if key in keyboard_cmds.keys():
      msg = keyboard_cmds[key]
      keyboard_control.identify_key(msg)
   else:
      msg = 80
      keyboard_control.identify_key(msg)
```

External controllers – Joystick Control

Joystick_control.py

```
from inputs import get_gamepad
# Importing the function to get input events from the gamepad
import math
import threading
# Importing threading module for multi-threading support

class XboxController(object):
    MAX_TRIG_VAL = math.pow(2, 8)    # Maximum value for triggers
    MAX_JOY_VAL = math.pow(2, 15)    # Maximum value for joysticks

    def __init__(self):
        # Initialize all button and joystick states to zero

        # Left stick
        self.LeftJoystickY = 0
        self.LeftJoystickX = 0

        # Right stick
        self.RightJoystickY = 0
        self.RightJoystickX = 0

        # Triggers
        self.LeftTrigger = 0
        self.RightTrigger = 0

        # Bumpers
        self.LeftBumper = 0
        self.RightBumper = 0

        # Face buttons
        self.A = 0
        self.X = 0
        self.Y = 0
        self.B = 0

        # Stick buttons
        self.LeftThumb = 0
        self.RightThumb = 0
```

External controllers – Joystick Control

```
        # Buttons
        self.Back = 0
        self.Start = 0

        # D-pad
        self.LeftDPad = 0
        self.RightDPad = 0
        self.UpDPad = 0
        self.DownDPad = 0

        # Thread to continuously monitor the controller
        self._monitor_thread = threading.Thread(target=self._monitor_controller, args=())
        self._monitor_thread.daemon = True
        self._monitor_thread.start()

    def read(self):
        # Return the current state of buttons and joysticks

        x = self.LeftJoystickX
        y = self.LeftJoystickY

        a = self.RightJoystickX
        b = self.RightJoystickY

        A = self.A
        B = self.B
        X = self.X
        Y = self.Y # b=1, x=2

        rb = self.RightBumper
        lb = self.LeftBumper

        rt = self.RightTrigger
        lt = self.LeftTrigger

        ld = self.LeftDPad
        rd = self.RightDPad
        ud = self.UpDPad
        dd = self.DownDPad
```

External controllers – Joystick Control

```
    # Return the values as a list
    return [x, y, a, b, A, B, X, Y, rb, lb, rt, lt, ld, rd, ud, dd]

  def _monitor_controller(self):
    # Continuously monitor the controller for input events
    while True:
      events = get_gamepad() # Get input events from the gamepad
      for event in events:
        # Update controller attributes based on input events

        if event.code == 'ABS_Y':
          self.LeftJoystickY = -(event.state / XboxController.MAX_JOY_VAL )
        # normalize joystick value between -1 and 1

        elif event.code == 'ABS_X':
          self.LeftJoystickX = event.state / XboxController.MAX_JOY_VAL
        # normalize joystick value between -1 and 1

        elif event.code == 'ABS_RY':
          self.RightJoystickY = -(event.state / XboxController.MAX_JOY_VAL)
        # normalize joystick value between -1 and 1

        elif event.code == 'ABS_RX':
          self.RightJoystickX = event.state / XboxController.MAX_JOY_VAL
        # normalize joystick value between -1 and 1

        elif event.code == 'ABS_Z':
          self.LeftTrigger = event.state / XboxController.MAX_TRIG_VAL
        # normalize joystick value between 0 and 1

        elif event.code == 'ABS_RZ':
          self.RightTrigger = event.state / XboxController.MAX_TRIG_VAL
        # normalize joystick value between 0 and 1

        elif event.code == 'BTN_TL':
                    self.LeftBumper = event.state
        elif event.code == 'BTN_TR':
                    self.RightBumper = event.state
```

External controllers – Joystick Control

```
            elif event.code == 'BTN_SOUTH':
                self.A = event.state
            elif event.code == 'BTN_NORTH':
                self.Y = event.state #previously switched with X
            elif event.code == 'BTN_WEST':
                self.X = event.state #previously switched with Y
            elif event.code == 'BTN_EAST':
                self.B = event.state

            elif event.code == 'BTN_THUMBL':
                self.LeftThumb = event.state
            elif event.code == 'BTN_THUMBR':
                self.RightThumb = event.state

            elif event.code == 'BTN_SELECT':
                self.Back = event.state
            elif event.code == 'BTN_START':
                self.Start = event.state

            elif event.code == 'BTN_TRIGGER_HAPPY1':
                self.LeftDPad = event.state
            elif event.code == 'BTN_TRIGGER_HAPPY2':
                self.RightDPad = event.state
                # print(event.state)
            elif event.code == 'BTN_TRIGGER_HAPPY3':
                self.UpDPad = event.state
            elif event.code == 'BTN_TRIGGER_HAPPY4':
                self.DownDPad = event.state

# Main block to create an instance of XboxController and continuously print its state
if __name__ == '__main__':
    joy = XboxController()
    while True:
        print(joy.read())
```

External controllers – Joystick Control

Joystick.py

```
import Joystick_control
# Importing the Joystick_controls module for Xbox controller input
from plutocontrol import pluto
# Importing the Pluto module for interfacing with the Pluto drone

# Initialize Xbox controller and Pluto drone objects
joy = Joystick_control.XboxController()
my_pluto = pluto()

# Function to map input range to output range
def mapping(x, inMin, inMax, outMin, outMax):
    x = (x - inMin) * (outMax - outMin) / (inMax - inMin) + outMin
    if (x < outMin):
      return int(outMin)
    elif (x > outMax):
      return int(outMax)
    else:
      return int(x)

# Main loop for continuously reading and processing controller input
while True:
   # Read input from the Xbox controller
    [x, y, a, b, A, B, X, Y, rb, lb, rt, lt, ld, rd, ud, dd] = joy.read()

     # Map controller input to drone controls
    my_pluto.rcThrottle = mapping(y, 1, -1, 1000, 2000)
    my_pluto.rcYaw = mapping(x, -1, 1, 1000, 2000)
    my_pluto.rcPitch = mapping(b, 1, -1, 1000, 2000)
    my_pluto.rcRoll = mapping(a, -1, 1, 1000, 2000)

    # Check button states for drone actions
    print(my_pluto.rcRoll)
```

External controllers – Joystick Control

```
# Check button states for drone actions
print(my_pluto.rcRoll)
if A:
 my_pluto.arm()  # Arm the drone
 # time.sleep(0.5)
 print("arming", A)
elif B:
 my_pluto.disarm()  # Disarm the drone
 print("disarming", B)
elif Y:
 my_pluto.take_off()  # Take off the drone
 # time.sleep(0.5)
 print("taken off", X)
elif X:
 my_pluto.land() # Land the drone
 my_pluto.disarm()
 print("landing", Y)
```

External controllers – Voice Control

We've developed a script that enables you to control your Pluto drone using your voice!

Download Pre-trained Voice Model:
Visit Vosk Models to download a pre-trained voice model.
Recommended models: vosk-model-small-en-us-0.15 or vosk-model-en-in-0.5.

Configure Voice Command Script:

Copy the path where you've stored your downloaded model.
Paste the path into the voice_cmd.py file located in the "voice" folder.

Run Voice Command Script:
Run the file: voice_cmd.py

Currently, the drone responds to the following voice commands:
"hello" to arm the drone.
"take off" to initiate takeoff.
"land" to initiate landing.

Feel free to customize these commands according to your preferences.

```
from vosk import Model, KaldiRecognizer
import pyaudio
import json
from plutocontrol import pluto

# Initialize Vosk model for speech recognition
model = Model(r"/vosk-model-small-en-us-0.15")
recognizer = KaldiRecognizer(model, 16000)

# Initialize PyAudio for microphone input
mic = pyaudio.PyAudio()  # Make sure your microphone is enabled and
working
stream = mic.open(format=pyaudio.paInt16, channels=1, rate=16000,
input=True, frames_per_buffer=8192)  # Increased buffer size
stream.start_stream()

# Initialize Pluto object for controlling the drone
my_pluto = pluto()
```

External controllers – Voice Control

```
# Continuously listen for speech input and recognize it
while True:
   data = stream.read(4096)

   # Process the received audio data for speech recognition
   if recognizer.AcceptWaveform(data):
      text = recognizer.Result()
      data = json.loads(text)
      recognized_text = data["text"]
      print("Recognized:", recognized_text)
            # Execute drone commands based on recognized speech

      if "start" in recognized_text:
         my_pluto.arm()
      elif "fly" in recognized_text:
         my_pluto.take_off()
      elif "stop" in recognized_text:
         my_pluto.disarm()
      elif "land" in recognized_text:
         my_pluto.land()
      elif "up" in recognized_text:
         my_pluto.increase_height()
      elif "down" in recognized_text:
         my_pluto.decrease_height()
      elif "forward" in recognized_text:
         my_pluto.forward()
      elif "back" in recognized_text:
         my_pluto.backward()
      elif "left" in recognized_text:
          my_pluto.left()
      elif "right" in recognized_text:
         my_pluto.right()
      elif "yes" in recognized_text:
         my_pluto.left_yaw()
      elif "no" in recognized_text:
         my_pluto.right_yaw()

   else:
      print("No speech detected or recognized.")
```

Working With Source Code

Guide to Predefined Functions: source code for Customizing and Modifying Drone Control Code:

This guide will help you understand the predefined functions in the drone control code, enabling you to create your own functions or modify existing ones. The code utilizes the MultiWii Serial Protocol (MSP) for communication with the drone.

The code establishes a connection with a drone using TCP/IP, constructs MultiWii Serial Protocol (MSP) packets, and sends them to the drone to control its behavior. It also defines a pluto class with various methods to control the drone's movements and states.

Table of Contents:

1. Constants and Configuration

The code defines several constants and configuration parameters used throughout the script.

- TRIM_MAX and TRIM_MIN: Maximum and minimum values for trim adjustments.
- TCP_IP and TCP_PORT: IP address and port for the TCP connection to the drone.
- MSP_HEADER_IN: Header for MSP packets.
- MSP_*: Constants representing different MSP commands.

2. Socket Connection Setup

A TCP socket connection is established to communicate with the drone.

```
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((TCP_IP, TCP_PORT))
```

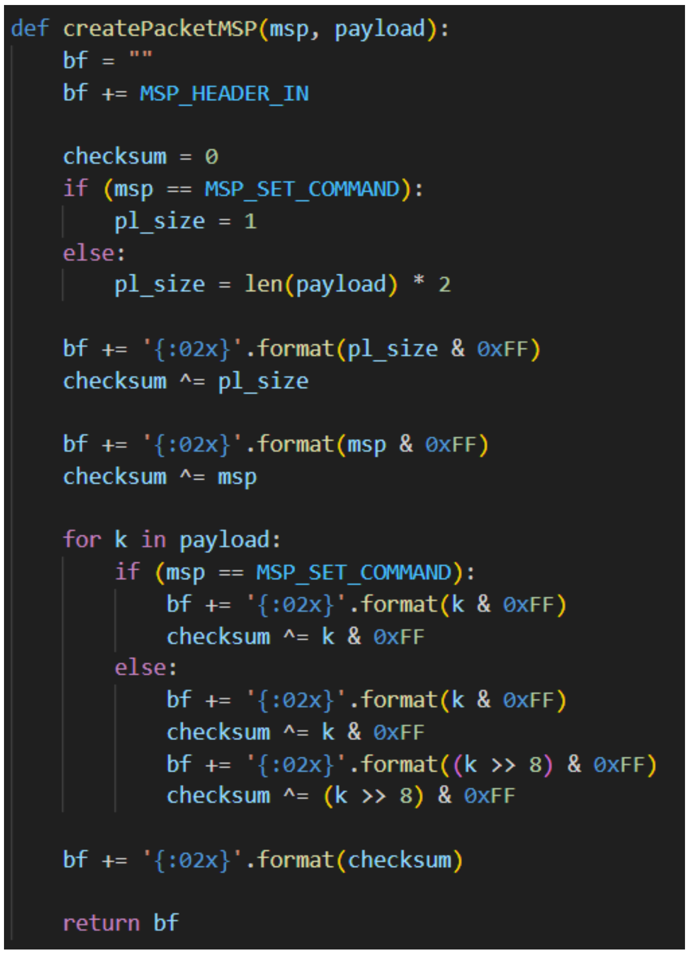
Working With Source Code

IP & PORT:

Without Camera:	IP: 192.168.4.1	PORT: 23
With Camera:	IP: 192.168.0.1	PORT: 9060

3. MSP Packet Creation:

The createPacketMSP function constructs MSP packets to be sent to the drone.
You can refer: https://create.dronaaviation.com/software/remote-programming/make-your-own for more information



```python
def createPacketMSP(msp, payload):
    bf = ""
    bf += MSP_HEADER_IN

    checksum = 0
    if (msp == MSP_SET_COMMAND):
        pl_size = 1
    else:
        pl_size = len(payload) * 2

    bf += '{:02x}'.format(pl_size & 0xFF)
    checksum ^= pl_size

    bf += '{:02x}'.format(msp & 0xFF)
    checksum ^= msp

    for k in payload:
        if (msp == MSP_SET_COMMAND):
            bf += '{:02x}'.format(k & 0xFF)
            checksum ^= k & 0xFF
        else:
            bf += '{:02x}'.format(k & 0xFF)
            checksum ^= k & 0xFF
            bf += '{:02x}'.format((k >> 8) & 0xFF)
            checksum ^= (k >> 8) & 0xFF

    bf += '{:02x}'.format(checksum)

    return bf
```

4. MSP Request Functions:

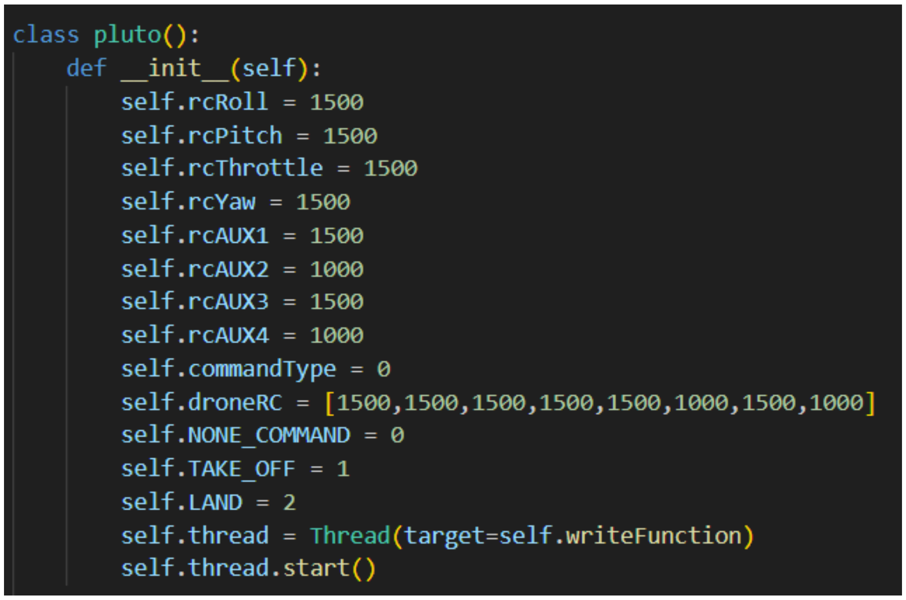
These functions send different MSP requests to the drone.

- sendRequestMSP(data): Sends raw MSP data to the drone.
- sendRequestMSP_SET_RAW_RC(channels): Sets the raw RC (remote control) values.
- sendRequestMSP_SET_COMMAND(commandType): Sends a specific command.
- sendRequestMSP_GET_DEBUG(requests): Requests debug information.
- sendRequestMSP_SET_ACC_TRIM(trim_roll, trim_pitch): Sets accelerometer trim values.
- sendRequestMSP_ACC_TRIM(): Requests accelerometer trim.
- sendRequestMSP_EEPROM_WRITE(): Writes to EEPROM.
- sendRequestMSP_SET_MOTOR(motor_speeds): Sets motor speeds.

5. Drone Control Class (pluto)

The pluto class contains methods to control the drone. It initializes with default RC values and starts a thread to continuously send control commands.

Constructor and Initialization



```
class pluto():
    def __init__(self):
        self.rcRoll = 1500
        self.rcPitch = 1500
        self.rcThrottle = 1500
        self.rcYaw = 1500
        self.rcAUX1 = 1500
        self.rcAUX2 = 1000
        self.rcAUX3 = 1500
        self.rcAUX4 = 1000
        self.commandType = 0
        self.droneRC = [1500,1500,1500,1500,1500,1000,1500,1000]
        self.NONE_COMMAND = 0
        self.TAKE_OFF = 1
        self.LAND = 2
        self.thread = Thread(target=self.writeFunction)
        self.thread.start()
```

Working With Source Code

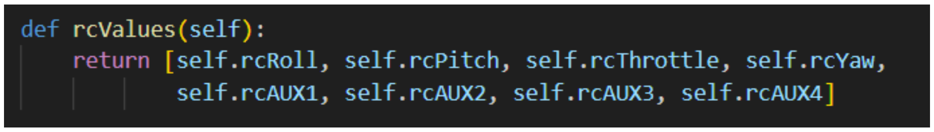
Basic Drone Commands:

These methods control the drone's basic movements.

- arm(): Arms the drone.
- box_arm(): Arms the drone in box mode.
- disarm(): Disarms the drone.
- forward(): Moves the drone forward.
- backward(): Moves the drone backward.
- left(): Rolls the drone to the left.
- right(): Rolls the drone to the right.
- left_yaw(): Yaws the drone to the left.
- right_yaw(): Yaws the drone to the right.
- reset(): Resets RC values.
- increase_height(): Increases drone's altitude.
- decrease_height(): Decreases drone's altitude.
- take_off(): Commands the drone to take off.
- land(): Commands the drone to land.

RC Values:

Returns the current RC values.



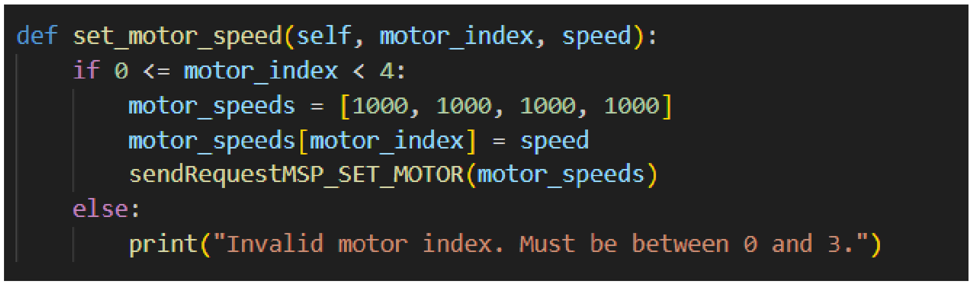
```python
def rcValues(self):
    return [self.rcRoll, self.rcPitch, self.rcThrottle, self.rcYaw,
            self.rcAUX1, self.rcAUX2, self.rcAUX3, self.rcAUX4]
```

Advanced Commands:

- trim_left_roll(): Trims the left roll.
- flip(): Commands the drone to flip.

Motor Speed Control

Sets the speed of a specific motor.

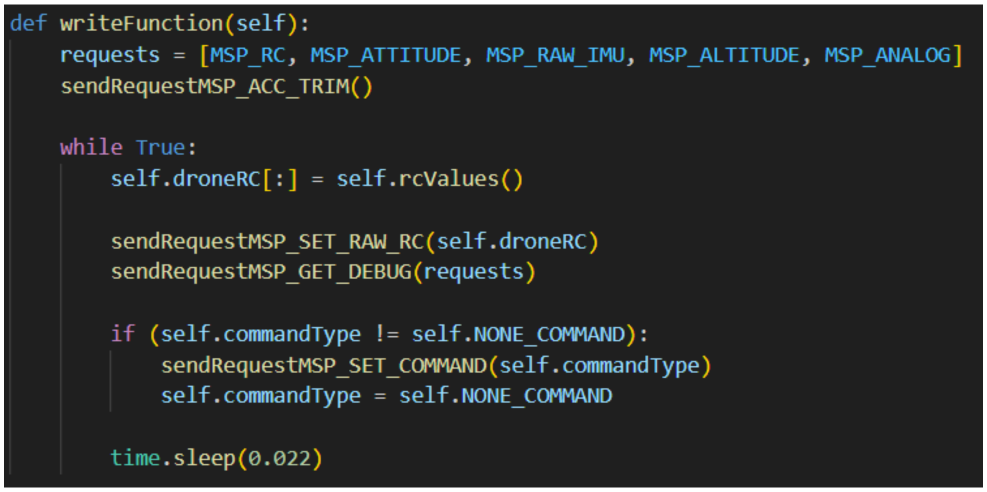


```python
def set_motor_speed(self, motor_index, speed):
    if 0 <= motor_index < 4:
        motor_speeds = [1000, 1000, 1000, 1000]
        motor_speeds[motor_index] = speed
        sendRequestMSP_SET_MOTOR(motor_speeds)
    else:
        print("Invalid motor index. Must be between 0 and 3.")
```

Working With Source Code

Continuous Communication Loop

The writeFunction method continuously sends control commands and requests debug information from the drone.



```
def writeFunction(self):
    requests = [MSP_RC, MSP_ATTITUDE, MSP_RAW_IMU, MSP_ALTITUDE, MSP_ANALOG]
    sendRequestMSP_ACC_TRIM()

    while True:
        self.droneRC[:] = self.rcValues()

        sendRequestMSP_SET_RAW_RC(self.droneRC)
        sendRequestMSP_GET_DEBUG(requests)

        if (self.commandType != self.NONE_COMMAND):
            sendRequestMSP_SET_COMMAND(self.commandType)
            self.commandType = self.NONE_COMMAND

        time.sleep(0.022)
```