9. Swarm Programming

The Tello Edu drone may be programmed in `swarm` mode; meaning, you may code behavior into multiple drones at once. The Tello Edu uses the Tello 2.0 SDK. The commands for version 2.0 are similar to 1.0, but there are many more commands. Here's a table listing the major commands.

Tello 2.0 Commands

Command	Description
command	Enter command mode
takeoff	Auto takeoff
land	Auto landing
streamon	Enable video stream
streamoff	Disable video stream
emergency	Stop motors immediately
stop	Hovers in the air
up `xx`	Fly upward [20, 500] cm
down `xx`	Fly downward [20, 500] cm
left `xx`	Fly left [20, 500] cm
right `xx`	Fly right [20, 500] cm
forward `xx`	Fly forward [20, 500] cm
back `xx`	Fly backward [20, 500] cm
cw `xx`	Rotate clockwise [1, 360] degrees
ccw `xx`	Rotate counter-clockwise [1, 360] degrees
flip `x`	Flip [l, r, f, b]
speed `x`	Set speed to [10, 100] cm/s
go `x` `y` `z` `speed`	Fly to x, y, z at speed
`x` `y` `z`	may be in the range [-500, 500]

Command	Description
	x y z may be in the range [-500, 500]
	speed is in the range [10, 100] cm/s
curve x1 y1 z1 x2 y2 z2 speed	Fly at a curve between the two given coordinates at sp
	x1 y1 z1 may be in the range [-500, 500]
	x2 y2 z2 may be in the range [-500, 500]
	speed is in the range [10, 60] cm/s
go x y z speed mid	Fly to x, y, z of mission pad at speed
	x y z may be in the range [-500, 500]
	speed is in the range [10, 100] cm/s
	mid is in the domain [m1, m2, m3, m4, m5, m6, m7, m8
curve x1 y1 z1 x2 y2 z2 speed mid	Fly at a curve between the two given coordinates of mi
	x1 y1 z1 may be in the range [-500, 500]
	x2 y2 z2 may be in the range [-500, 500]
	speed is in the range [10, 60] cm/s
	mid is in the domain [m1, m2, m3, m4, m5, m6, m7, m8
jump x y z speed yaw mid1 mid2	Fly to x, y, z of mission pad 1 and recognize coordinates
	x y z may be in the range [-500, 500]
	speed is in the range [10, 100] cm/s
	mid is in the domain [m1, m2, m3, m4, m5, m6, m7, m8
wifi ssid pass	Set WiFi password
mon	Enable mission pad detection
moff	Disable mission pad detection
mdirection x	Enable mission pad detection
	x = 0, downward detection only
	x = 1, forward detection only
	x = 2, downward and forward detection
ap ssid pass	Set Tello to station mode and connect to new access pc
speed?	Get current speed
battery?	Get current battery percentage
time?	Get current flight time

Command	Description
wifi?	Get WiFi SNR
sdk?	Get the SDK version
sn?	Get the serial number

9.1. Dependencies

To start using Python to manipulate the Tello Swarm, make sure you install the following packages `netifaces` and `netaddr`.

```
conda install -y -c conda-forge netifaces netaddr
```

Make sure you are using Python 3.7 or higher. The original code requires Python 2.7, but we have re-written the code for Python 3.7 and heavily refactored it to be easier to maintain and read.

9.2. Set Tello modes

Each Tello Edu can exist in `AP Mode` or `Station Mode`.

- `AP Mode` or `Access Point Mode` is when the Tello becomes a client to a router.
- `Station Mode` is when the Tello acts like a router.

Only when a Tello Edu is set to `AP Mode` will you be able to use Python to do swarm programming. The script `set-ap-mode.py` will help you set the Tello Edu to `AP Mode`. To reset the Tello Edu back to `Station Mode`, turn on the drone and then hold the power button for 5 seconds. Below is an example usage of the script; you will need to provide the `SSID` and password of the router to the program. Additionally, make sure your router supports the 2.4 GHz bandwidth, as the drone will not connect to the 5.0 GHz bandwidth.

```
python set-ap-mode.py -s [SSID] -p [PASSWORD]
```

The code for `set-ap-mode.py` is listed below.

```python
import socket
import argparse
import sys

def get_socket():
    """
    Gets a socket.
    :return: Socket.
    """
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    s.bind(('', 8889))

    return s

def set_ap(ssid, password, address):
    """
    A Function to set tello in Access Point (AP) mode.

    :param ssid: The SSID of the network (e.g. name of the Wi-Fi).
    :param password: The password of the network.
    :param address: Tello IP.
    :return: None.
    """
    s = get_socket()

    cmd = 'command'
    print(f'sending cmd {cmd}')
    s.sendto(cmd.encode('utf-8'), address)

    response, ip = s.recvfrom(100)
    print(f'from {ip}: {response}')

    cmd = f'ap {ssid} {password}'
    print(f'sending cmd {cmd}')
    s.sendto(cmd.encode('utf-8'), address)

    response, ip = s.recvfrom(100)
    print(f'from {ip}: {response}')

def parse_args(args):
    """
    Parses arguments.
    :param args: Arguments.
    :return: Arguments.
    """
    parser = argparse.ArgumentParser('set-ap-mode.py',
                epilog='One-Off Coder http://www.oneoffcoder.com')

    parser.add_argument('-s', '--ssid', help='SSID', required=True)
    parser.add_argument('-p', '--pwd', help='password', required=True)
    parser.add_argument('--ip', help='Tello IP', default='192.168.10.1', required=False)
    parser.add_argument('--port', help='Tello port', default=8889, type=int, required=False)
    parser.add_argument('--version', action='version', version='%(prog)s v0.0.1')

    return parser.parse_args(args)

if __name__ == '__main__':
    args = parse_args(sys.argv[1:])
    ssid = args.ssid
```

```
60      pwd = args.pwd
61      tello_address = (args.ip, args.port)
62
63      set_ap(ssid, pwd, tello_address)
```

9.3. Python programming

There are 6 main Python classes created to manipulate the drones, and they are listed below.

Python Classes

ID	Class Name	Purpose
1	Stats	Collect statistics
2	SubnetInfo	Stores subnet information
3	Tello	Models a Tello EDU (or drone)
4	TelloManager	Manages connections to drones
5	SwarmUtil	Utility class to help swarm programming
6	Swarm	Models a swarm of drones

All these classes are brought together by a single program `planned-flight.py`, which is the entry point where your pre-defined commands are sent to the swarm. The `planned-flight.py` program is very simple and looks like the following.

```
import sys
import argparse
from swarm import *

def parse_args(args):
    """
    Parses arguments.
    :param args: Arguments.
    :return: Arguments.
    """
    parser = argparse.ArgumentParser('planned-flight.py',
                    epilog='One-Off Coder http://www.oneoffcoder.com')

    parser.add_argument('-f', '--file', help='Command text file', required=True)
    parser.add_argument('--version', action='version', version='%(prog)s v0.0.1')

    return parser.parse_args(args)

if __name__ == '__main__':
    args = parse_args(sys.argv[1:])
    fpath = args.file

    swarm = Swarm(fpath)
    swarm.start()
```

As you can see, `planned-flight.py` takes in a file path as input. The file pointed to by the file path is simply a text file of the commands supported by the SDK. An example of the command file is as follows.

```
scan 1
battery_check 20
correct_ip
1=0TQZGANED0021X
1>takeoff
sync 1
1>land
```

You may then execute the program as follows.

```
python planned-flight.py -f cmds-01.txt
```

Here's the `Stats` class.

```
class Stats(object):
    """
    Statistics
    """

    def __init__(self, command, id):
        """
        Ctor.
        :param command: Command.
        :param id: ID.
        """
        self.command = command
        self.response = None
        self.id = id

        self.start_time = datetime.now()
        self.end_time = None
        self.duration = None
        self.drone_ip = None

    def add_response(self, response, ip):
        """
        Adds a response.
        :param response: Response.
        :param ip: IP address.
        :return: None.
        """
        if self.response == None:
            self.response = response
            self.end_time = datetime.now()
            self.duration = self.get_duration()
            self.drone_ip = ip

    def get_duration(self):
        """
        Gets the duration.
        :return: Duration (seconds).
        """
        diff = self.end_time - self.start_time
        return diff.total_seconds()

    def print_stats(self):
        """
        Prints statistics.
        :return: None.
        """
        print(self.get_stats())

    def got_response(self):
        """
        Checks if response was received.
        :return: A boolean indicating if response was received.
        """
        return False if self.response is None else True

    def get_stats(self):
        """
        Gets the statistics.
        :return: Statistics.
```

```
        """
        return {
            'id': self.id,
            'command': self.command,
            'response': self.response,
            'start_time': self.start_time,
            'end_time': self.end_time,
            'duration': self.duration
        }

    def get_stats_delimited(self):
        stats = self.get_stats()
        keys = ['id', 'command', 'response', 'start_time', 'end_time', 'duration']
        vals = [f'{k}={stats[k]}' for k in keys]
        vals = ', '.join(vals)
        return vals

    def __repr__(self):
        return self.get_stats_delimited()
```

Here's the `SubnetInfo` class.

```
class SubnetInfo(object):
    """
    Subnet information.
    """

    def __init__(self, ip, network, netmask):
        """
        Ctor.
        :param ip: IP.
        :param network: Network.
        :param netmask: Netmask.
        """
        self.ip = ip
        self.network = network
        self.netmask = netmask

    def __repr__(self):
        return f'{self.network} | {self.netmask} | {self.ip}'

    def get_ips(self):
        """
        Gets all the possible IP addresses in the subnet.
        :return: List of IPs.
        """
        def get_quad(ip):
            """
            Gets the third quad.
            :param ip: IP.
            :return: Third quad.
            """
            quads = str(ip).split('.')
            quad = quads[3]
            return quad

        def is_valid(ip):
            """
            Checks if IP is valid.
            :return: A boolean indicating if IP is valid.
            """
            quad = get_quad(ip)
            result = False if quad == '0' or quad == '255' else True

            if result:
                if str(ip) == self.ip:
                    result = False

            return result

        ip_network = IPNetwork(f'{self.network}/{self.netmask}')

        return [str(ip) for ip in ip_network if is_valid(ip)]

    @staticmethod
    def flatten(infos):
        return list(itertools.chain.from_iterable(infos))
```

Here's the `Tello` class.

```python
class Tello(object):
    """
    A wrapper class to interact with Tello.
    Communication with Tello is handled by TelloManager.
    """
    def __init__(self, tello_ip, tello_manager):
        """
        Ctor.
        :param tello_ip: Tello IP.
        :param tello_manager: Tello Manager.
        """
        self.tello_ip = tello_ip
        self.tello_manager = tello_manager

    def send_command(self, command):
        """
        Sends a command.
        :param command: Command.
        :return: None.
        """
        return self.tello_manager.send_command(command, self.tello_ip)

    def __repr__(self):
        return f'TELLO@{self.tello_ip}'
```

Here's the `TelloManager` class.

```
class TelloManager(object):
    """
    Tello Manager.
    """

    def __init__(self):
        """
        Ctor.
        """
        self.local_ip = ''
        self.local_port = 8889
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        self.socket.bind((self.local_ip, self.local_port))

        # thread for receiving cmd ack
        self.receive_thread = threading.Thread(target=self._receive_thread)
        self.receive_thread.daemon = True
        self.receive_thread.start()

        self.tello_ip_list = []
        self.tello_list = []
        self.log = defaultdict(list)

        self.COMMAND_TIME_OUT = 20.0

        self.last_response_index = {}
        self.str_cmd_index = {}

    def find_avaliable_tello(self, num):
        """
        Find Tellos.
        :param num: Number of Tellos to search.
        :return: None
        """
        possible_ips = self.get_possible_ips()

        print(f'[SEARCHING], Searching for {num} from {len(possible_ips)} possible IP
addresses')

        iters = 0

        while len(self.tello_ip_list) < num:
            print(f'[SEARCHING], Trying to find Tellos, number of tries = {iters + 1}')

            # delete already found Tello
            for tello_ip in self.tello_ip_list:
                if tello_ip in possible_ips:
                    possible_ips.remove(tello_ip)

            # skip server itself
            for ip in possible_ips:
                cmd_id = len(self.log[ip])
                self.log[ip].append(Stats('command', cmd_id))

                # print(f'{iters}: sending command to {ip}:8889')

                try:
                    self.socket.sendto(b'command', (ip, 8889))
                except:
```

```
                    print(f'{iters}: ERROR: {ip}:8889')
                    pass

            iters = iters + 1
            time.sleep(5)

        # filter out non-tello addresses in log
        temp = defaultdict(list)
        for ip in self.tello_ip_list:
            temp[ip] = self.log[ip]
        self.log = temp

    def get_possible_ips(self):
        """
        Gets all the possible IP addresses for subnets that the computer is a part of.
        :return: List of IP addresses.
        """
        infos = self.get_subnets()
        ips = SubnetInfo.flatten([info.get_ips() for info in infos])
        ips = list(filter(lambda ip: ip.startswith('192.168.3.'), ips))
        return ips

    def get_subnets(self):
        """
        Gets all subnet information.

        :return: List of subnet information.
        """
        infos = []

        for iface in netifaces.interfaces():
            addrs = netifaces.ifaddresses(iface)

            if socket.AF_INET not in addrs:
                continue

            # Get ipv4 stuff
            ipinfo = addrs[socket.AF_INET][0]
            address, netmask = ipinfo['addr'], ipinfo['netmask']

            # limit range of search. This will work for router subnets
            if netmask != '255.255.255.0':
                continue

            # Create ip object and get
            cidr = netaddr.IPNetwork(f'{address}/{netmask}')
            network = cidr.network

            info = SubnetInfo(address, network, netmask)
            infos.append(info)

        return infos

    def get_tello_list(self):
        return self.tello_list

    def send_command(self, command, ip):
        """
        Sends a command to the IP address. Will be blocked until the last command receives an 'OK'.
```

```
        If the command fails (either b/c time out or error),  will try to resend the
command.

        :param command: Command.
        :param ip: Tello IP.
        :return: Response.
        """
        #global cmd
        command_sof_1 = ord(command[0])
        command_sof_2 = ord(command[1])

        if command_sof_1 == 0x52 and command_sof_2 == 0x65:
            multi_cmd_send_flag = True
        else :
            multi_cmd_send_flag = False

        if multi_cmd_send_flag == True:
            self.str_cmd_index[ip] = self.str_cmd_index[ip] + 1
            for num in range(1,5):
                str_cmd_index_h = self.str_cmd_index[ip] / 128 + 1
                str_cmd_index_l = self.str_cmd_index[ip] % 128
                if str_cmd_index_l == 0:
                    str_cmd_index_l = str_cmd_index_l + 2
                cmd_sof = [0x52, 0x65, str_cmd_index_h, str_cmd_index_l, 0x01, num + 1,
0x20]
                cmd_sof_str = str(bytearray(cmd_sof))
                cmd = cmd_sof_str + command[3:]
                self.socket.sendto(cmd.encode('utf-8'), (ip, 8889))

            print(f'[MULTI_COMMAND], IP={ip}, COMMAND={command[3:]}')
            real_command = command[3:]
        else:
            self.socket.sendto(command.encode('utf-8'), (ip, 8889))
            print(f'[SINGLE_COMMAND] IP={ip}, COMMAND={command}')
            real_command = command

        self.log[ip].append(Stats(real_command, len(self.log[ip])))
        start = time.time()

        while not self.log[ip][-1].got_response():
            now = time.time()
            diff = now - start
            if diff > self.COMMAND_TIME_OUT:
                print(f'[NO_RESPONSE] Max timeout exceeded for command: {real_command}')
                return

    def _receive_thread(self):
        """
        Listen to responses from the Tello.
        Runs as a thread, sets self.response to whatever the Tello last returned.

        :return: None.
        """
        while True:
            try:
                response, ip = self.socket.recvfrom(1024)
                response = response.decode('utf-8')
                self.response = response

                ip = ''.join(str(ip[0]))
```

```
176
177                 if self.response.upper() == 'OK' and ip not in self.tello_ip_list:
178                     self.tello_ip_list.append(ip)
179                     self.last_response_index[ip] = 100
180                     self.tello_list.append(Tello(ip, self))
181                     self.str_cmd_index[ip] = 1
182
183                 response_sof_part1 = ord(self.response[0])
184                 response_sof_part2 = ord(self.response[1])
185
186                 if response_sof_part1 == 0x52 and response_sof_part2 == 0x65:
187                     response_index = ord(self.response[3])
188
189                     if response_index != self.last_response_index[ip]:
190                         print(f'[MULTI_RESPONSE], IP={ip}, RESPONSE={self.response[7:]}')
191                         self.log[ip][-1].add_response(self.response[7:], ip)
192                     self.last_response_index[ip] = response_index
193                 else:
194                     # print(f'[SINGLE_RESPONSE], IP={ip}, RESPONSE={self.response}')
195                     self.log[ip][-1].add_response(self.response, ip)
196
197             except socket.error as exc:
198                 # swallow exception
199                 # print "[Exception_Error]Caught exception socket.error : %s\n" % exc
200                 pass
201
202     def get_log(self):
203         """
204         Get all logs.
205         :return: Dictionary of logs.
206         """
207         return self.log
208
209     def get_last_logs(self):
210         """
211         Gets the last logs.
212         :return: List of last logs.
213         """
214         return [log[-1] for log in self.log.values()]
```

Here's the `SwarmUtil` class.

```python
class SwarmUtil(object):
    """
    Swarm utility class.
    """

    @staticmethod
    def create_execution_pools(num):
        """
        Creates execution pools.

        :param num: Number of execution pools to create.
        :return: List of Queues.
        """
        return [queue.Queue() for x in range(num)]


    @staticmethod
    def drone_handler(tello, queue):
        """
        Drone handler.

        :param tello: Tello.
        :param queue: Queue.
        :return: None.
        """
        while True:
            while queue.empty():
                pass
            command = queue.get()
            tello.send_command(command)


    @staticmethod
    def all_queue_empty(pools):
        """
        Checks if all queues are empty.

        :param pools: List of Queues.
        :return: Boolean indicating if all queues are empty.
        """
        for queue in pools:
            if not queue.empty():
                return False
        return True


    @staticmethod
    def all_got_response(manager):
        """
        Checks if all responses are received.

        :param manager: TelloManager.
        :return: A boolean indicating if all responses are received.
        """
        for log in manager.get_last_logs():
            if not log.got_response():
                return False
        return True

```

```
60
61      @staticmethod
62      def create_dir(dpath):
63          """
64          Creates a directory if it does not exists.
65
66          :param dpath: Directory path.
67          :return: None.
68          """
69          if not os.path.exists(dpath):
70              with suppress(Exception):
71                  os.makedirs(dpath)
72
73      @staticmethod
74      def save_log(manager):
75          """
76          Saves the logs into a file in the ./log directory.
77
78          :param manager: TelloManager.
79          :return: None.
80          """
81          dpath = './log'
82          SwarmUtil.create_dir(dpath)
83
84          start_time = str(time.strftime("%Y-%m-%d_%H-%M-%S", time.localtime(time.time())))
85          fpath = f'{dpath}/{start_time}.txt'
86
87          with open(fpath, 'w') as out:
88              log = manager.get_log()
89              for cnt, stats in enumerate(log.values()):
90                  out.write(f'------\nDrone: {cnt + 1}\n')
91
92                  s = [stat.get_stats_delimited() for stat in stats]
93                  s = '\n'.join(s)
94
95                  out.write(f'{s}\n')
96
97          print(f'[LOG] Saved log files to {fpath}')
98
99
100     @staticmethod
101     def check_timeout(start_time, end_time, timeout):
102         """
103         Checks if the duration between the end and start times
104         is larger than the specified timeout.
105
106         :param start_time: Start time.
107         :param end_time: End time.
108         :param timeout: Timeout threshold.
109         :return: A boolean indicating if the duration is larger than the specified timeout
threshold.
110         """
111         diff = end_time - start_time
112         time.sleep(0.1)
113         return diff > timeout
```

Here's the `Swarm` class.

```
class Swarm(object):
    """
    Tello Edu swarm.
    """

    def __init__(self, fpath):
        """
        Ctor.

        :param fpath: Path to command text file.
        """
        self.fpath = fpath
        self.commands = self._get_commands(fpath)
        self.manager = TelloManager()
        self.tellos = []
        self.pools = []
        self.sn2ip = {
            '0TQZGANED0021X': '192.168.3.101',
            '0TQZGANED0020C': '192.168.3.103',
            '0TQZGANED0023H': '192.168.3.104'
        }
        self.id2sn = {
            0: '0TQZGANED0021X',
            1: '0TQZGANED0020C',
            2: '0TQZGANED0023H'
        }
        self.ip2id = {
            '192.168.3.101': 0,
            '192.168.3.103': 1,
            '192.168.3.104': 2
        }

    def start(self):
        """
        Main loop. Starts the swarm.

        :return: None.
        """
        def is_invalid_command(command):
            if command is None:
                return True
            c = command.strip()
            if len(c) == 0:
                return True
            if c == '':
                return True
            if c == '\n':
                return True
            return False

        try:
            for command in self.commands:
                if is_invalid_command(command):
                    continue

                command = command.rstrip()

                if '//' in command:
                    self._handle_comments(command)
```

```
                elif 'scan' in command:
                    self._handle_scan(command)
                elif '>' in command:
                    self._handle_gte(command)
                elif 'battery_check' in command:
                    self._handle_battery_check(command)
                elif 'delay' in command:
                    self._handle_delay(command)
                elif 'correct_ip' in command:
                    self._handle_correct_ip(command)
                elif '=' in command:
                    self._handle_eq(command)
                elif 'sync' in command:
                    self._handle_sync(command)

            self._wait_for_all()
        except KeyboardInterrupt as ki:
            self._handle_keyboard_interrupt()
        except Exception as e:
            self._handle_exception(e)
            traceback.print_exc()
        finally:
            SwarmUtil.save_log(self.manager)

    def _wait_for_all(self):
        """
        Waits for all queues to be empty and for all responses
        to be received.

        :return: None.
        """
        while not SwarmUtil.all_queue_empty(self.pools):
            time.sleep(0.5)

        time.sleep(1)

        while not SwarmUtil.all_got_response(self.manager):
            time.sleep(0.5)

    def _get_commands(self, fpath):
        """
        Gets the commands.

        :param fpath: Command file path.
        :return: List of commands.
        """
        with open(fpath, 'r') as f:
            return f.readlines()

    def _handle_comments(self, command):
        """
        Handles comments.

        :param command: Command.
        :return: None.
        """
        print(f'[COMMENT] {command}')

    def _handle_scan(self, command):
        """
```

```
        Handles scan.

        :param command: Command.
        :return: None.
        """
        n_tellos = int(command.partition('scan')[2])

        self.manager.find_avaliable_tello(n_tellos)
        self.tellos = self.manager.get_tello_list()
        self.pools = SwarmUtil.create_execution_pools(n_tellos)

        for x, (tello, pool) in enumerate(zip(self.tellos, self.pools)):
            self.ip2id[tello.tello_ip] = x

            t = Thread(target=SwarmUtil.drone_handler, args=(tello, pool))
            t.daemon = True
            t.start()

            print(f'[SCAN] IP = {tello.tello_ip}, ID = {x}')

    def _handle_gte(self, command):
        """
        Handles gte or >.

        :param command: Command.
        :return: None.
        """
        id_list = []
        id = command.partition('>')[0]

        if id == '*':
            id_list = [t for t in range(len(self.tellos))]
        else:
            id_list.append(int(id)-1)

        action = str(command.partition('>')[2])

        for tello_id in id_list:
            sn = self.id2sn[tello_id]
            ip = self.sn2ip[sn]
            id = self.ip2id[ip]

            self.pools[id].put(action)
            print(f'[ACTION] SN = {sn}, IP = {ip}, ID = {id}, ACTION = {action}')

    def _handle_battery_check(self, command):
        """
        Handles battery check. Raises exception if any drone has
        battery life lower than specified threshold in the command.

        :param command: Command.
        :return: None.
        """
        threshold = int(command.partition('battery_check')[2])
        for queue in self.pools:
            queue.put('battery?')

        self._wait_for_all()

        is_low = False
```

```
180
181         for log in self.manager.get_last_logs():
182             battery = int(log.response)
183             drone_ip = log.drone_ip
184
185             print(f'[BATTERY] IP = {drone_ip}, LIFE = {battery}%')
186
187             if battery < threshold:
188                 is_low = True
189
190         if is_low:
191             raise Exception('Battery check failed!')
192         else:
193             print('[BATTERY] Passed battery check')
194
195     def _handle_delay(self, command):
196         """
197         Handles delay.
198
199         :param command: Command.
200         :return: None.
201         """
202         delay_time = float(command.partition('delay')[2])
203         print (f'[DELAY] Start Delay for {delay_time} second')
204         time.sleep(delay_time)
205
206     def _handle_correct_ip(self, command):
207         """
208         Handles correction of IPs.
209
210         :param command: Command.
211         :return: None.
212         """
213         for queue in self.pools:
214             queue.put('sn?')
215
216         self._wait_for_all()
217
218         for log in self.manager.get_last_logs():
219             sn = str(log.response)
220             tello_ip = str(log.drone_ip)
221             self.sn2ip[sn] = tello_ip
222
223             print(f'[CORRECT_IP] SN = {sn}, IP = {tello_ip}')
224
225     def _handle_eq(self, command):
226         """
227         Handles assignments of IDs to serial numbers.
228
229         :param command: Command.
230         :return: None.
231         """
232         id = int(command.partition('=')[0])
233         sn = command.partition('=')[2]
234         ip = self.sn2ip[sn]
235
236         self.id2sn[id-1] = sn
237
238         print(f'[IP_SN_ID] IP = {ip}, SN = {sn}, ID = {id}')
239
```

```
240         def _handle_sync(self, command):
241             """
242             Handles synchronization.
243 
244             :param command: Command.
245             :return: None.
246             """
247             timeout = float(command.partition('sync')[2])
248             print(f'[SYNC] Sync for {timeout} seconds')
249 
250             time.sleep(1)
251 
252             try:
253                 start = time.time()
254 
255                 while not SwarmUtil.all_queue_empty(self.pools):
256                     now = time.time()
257                     if SwarmUtil.check_timeout(start, now, timeout):
258                         raise RuntimeError('Sync failed since all queues were not empty!')
259 
260                 print('[SYNC] All queues empty and all commands sent')
261 
262                 while not SwarmUtil.all_got_response(self.manager):
263                     now = time.time()
264                     if SwarmUtil.check_timeout(start, now, timeout):
265                         raise RuntimeError('Sync failed since all responses were not
received!')
266 
267                 print('[SYNC] All response received')
268             except RuntimeError:
269                 print('[SYNC] Failed to sync; timeout exceeded')
270 
271         def _handle_keyboard_interrupt(self):
272             """
273             Handles keyboard interrupt.
274 
275             :param command: Command.
276             :return: None.
277             """
278             print('[QUIT_ALL], KeyboardInterrupt. Sending land to all drones')
```

9.4. Download

The files to program your Tello swarm may be downloaded. Note that you will have to modify the command files for your own drones (e.g. serial numbers).

- planned-flight.py
- tello.py
- swarm.py
- cmds-01.txt
- cmds-02.txt
- cmds-03.txt

- cmds-04.txt