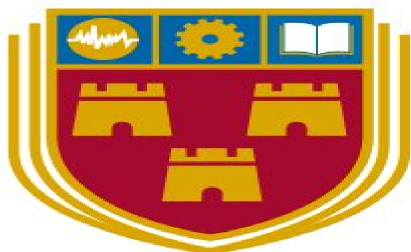
Drone Air Traffic Control System

Software to Coordinate Automated Drones, Safely



INSTITUTE of TECHNOLOGY CARLOW

Institiúid Teicneolaíochta Cheatharlach

Ronan Donohue - c00208501

System Requirements:

Dependencies:

This section covers the dependencies required to run Drone Air Traffic Control System.

Anaconda:
This covers most of the Qt/QML stuff. The most recent version they use is Qt.5.9.
If you want to be compatible with DATCS, you'll need either the latest version of Anaconda installed or the older version of Qt 5.9 (Latest version of Qt is 5.12 but isn't compatible with Python 3.7) How to install: https://docs.anaconda.com/anaconda/install/linux/

Mysql -
```
$ sudo apt-get update
$ sudo apt-get install mysql-server
```

pymysql -
```
$ python3 -m pip install PyMySQL
```
If there are any difficulties with pymysql, consult;
https://pymysql.readthedocs.io/en/latest/user/installation.html

Pyparrot;
```
$ pip install pyparrot
```
If there are any difficulties, consult;
https://pyparrot.readthedocs.io/en/latest/installation.html

Running the code:

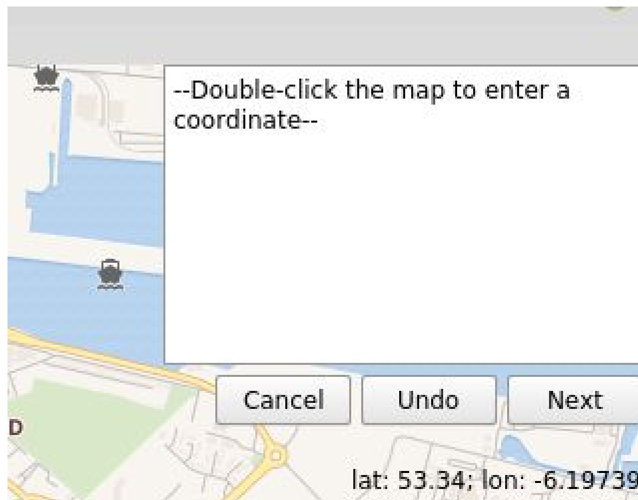
To run the code, navigate to the directory where you installed the Drone Air Traffic Control System and from the terminal run;
```
$ python DATCS.py
```

This will begin the program.

How to use the software:

Creating a Flight Plan

To create a flight plan, select *Flight Plan ->New Flight Plan*. To place a coordinate for your flight plan, double-click the screen. To add more points, continue to double-click on the map where you desire to go to. If you make a mistake, you can click *Undo* or if you wish to return, click *Cancel*.

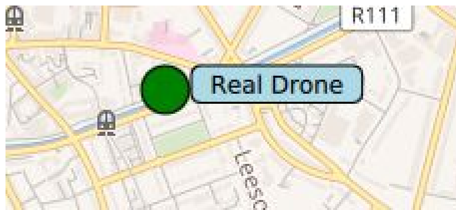


Creating a Simulated Drone

To create a simulated drone, *Drone* -> *New Sim Drone*. To place your simulated drone, select a point on the map and double click. To assign a flight plan to your drone, right-click the drone icon and select *'Assign Flight Plan'*. Once the flight plan has been assigned, to have the drone run the flight plan you must click on the drone.

Creating a Real Drone

To create a real drone, first, make sure you are connected to the drone's wifi and click *Drone* -> *New Real Drone* . After a short while, the drone will display on the map as a Green circle. You can right-click the drone to perform similar actions to the simulated drone.



Source Code:

The following pages are the full code listings for the Drone Air Traffic Control System.

```
#Author: Ronan Donohue
#Number: c00208501

__license__ = "MIT"
__revision__ = "AirTrafficController.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'

import time
import geocoder
from threading import Thread
import subprocess

from PyQt5.QtCore import QObject, pyqtSignal, pyqtSlot, pyqtProperty, QVariant, QTim
er
from PyQt5.QtPositioning import QGeoCoordinate

from FlightPlan import FlightPlan
from DroneController import DroneController
from Loader import Loader
from CollisionPredictor import CollisionPredictor


class AirTrafficController(QObject):
    """
    This class is responsible for creating simulated drones and flight plans.

    PyQt Signals emit infomation to the QML view.
    PyQtSlots receive data from the QML view.
    PyQtProperties exist in both model and view.

    Each DroneController is a pilot of a SimulatedDrone or a DATCS_Bebop drone.
    Each DroneController will be assigned a FlightPlan.
    FlightPlan data is brought in via the Loader class.
    While each DroneController is undertaking a FlightPlan, their information
    is passed to a CollisionPredictor object, which determines if any of the drones
are
    in immediate danger of a collision.
    """

    ##All PyQtSignals
    locationChanged = pyqtSignal(QGeoCoordinate)
    listFPNames = pyqtSignal(QVariant, arguments=['flightPlanNames'])
    moveDrone = pyqtSignal(str, QGeoCoordinate, arguments=['thisDroneName','currentL
ocation'])
    sendDroneInfo = pyqtSignal(str, arguments=['requestingDrone'])
    stopDrone = pyqtSignal(str, arguments=['stopDroneName'])
    plotDroneRouteViz = pyqtSignal(list, str, arguments=['flight_plan_coordinates',
'flight_plan_name'])
    removeDronePath = pyqtSignal(str, arguments=['path_name'])
    updateATCConsole = pyqtSignal(str, arguments=['update'])
    movingHome = pyqtSignal(str, arguments=['offhome'])
    arrivedHome = pyqtSignal(str, str, arguments=['home', 'homeDrone'])
    flightPlanAlreadyAssigned = pyqtSignal(str, arguments=['assignedFPName'])
    flightPlanFinished = pyqtSignal(str, arguments=['droneFinishedFP'])

    def __init__(self, parent=None):
        """
        Initializes an AirTrafficController instance,
        Dictionaries are used to hold information about
        drones, their current status, the paths they must traverse
        and information about each flight plan.

        A QTimer polls the active drones every second for information.
        """
        super().__init__(parent)
        self._running = True

        self.drones = {}
        self.drone_status = {}
        self.drone_paths = {}
        self.flight_plans = {}
```

```
        self.poll_timer = QTimer()

        latlng = geocoder.ip('me').latlng
        if latlng ==  None:
            # we've connected to the drone in this case, so we're offline technicall
y...
            # we'll use the college coordinates as this is an IT Carlow project
            # to set the center of the map.
            latlng = [52.8408, -6.9261]
        self._location = QGeoCoordinate(latlng[0], latlng[1], 0.0)
        self.cp = CollisionPredictor(self._location)
        self.loader = Loader()

        self.poll_timer.timeout.connect(self.request_drone_info)
        self.poll_timer.start(1000)


###############################################################################
    """
    This section of the code contains most of the functions relating to
    PyQt5/QML singal/slot/property functionality
    """


    @pyqtSlot()
    def list_flight_plans(self):
        """
        This function gets all the flight plan names from the Loader
        and passes them into the QML View.
        """
        names = self.loader.get_flight_plan_names()
        self.listFPNames.emit(names)



    @pyqtSlot(str)
    def create_flight_plan(self, name):
        """
        This function creates a FlightPlan object with param name.
        If the param name is the name of an existing flight plan, that
        flight plan will be loaded into the AirTrafficController. If the
        param name is the name of a new FlightPlan, a new FlightPlan instance will b
e
        created.

        :param name: str, representing the name of the FlightPlan to be created
        """
        fp = FlightPlan(name)
        self.flight_plans[name]= fp


    @pyqtSlot(str, str, str, str)
    def append_flight_plan_coordinates(self, name, lat, lng, alt):
        """
        Appends a coordinate to the FlightPlan with name param name.
        Param lat, lng and alt are converted to float representation prior
        to storage.

        :param name: str, representing the FlightPlan name
        :param lat: str, representing the coordinate lat to be added
        :param lng: str, representing the coordinate lng to be added
        :param alt: str, representing the coordinate alt to be added
        """
        c_list = []
        c_list.append(float(lat))
        c_list.append(float(lng))
        c_list.append(float(alt))
        c_list.append(0)

        self.flight_plans[name].append_new_coordinate(c_list)
```

```
    @pyqtSlot()
    def get_location(self):
        """
        PyQtSlot assigned to AirTrafficController property self._location
        Returns self._location
        """
        return self._location



    def set_location(self, coordinate):
        """
        PyQt setter function for property self._location

        :param coordinate: new value for self._location
        """
        if self._location != coordinate:
            self._location = coordinate
            self.locationChanged.emit(self._location)


    #pyqt Property
    location = pyqtProperty(QGeoCoordinate,
            fget=get_location,
            fset=set_location,
            notify=locationChanged)



    @pyqtSlot(str, str, str, str)
    def add_drone(self, name, lat, lng, which_type):
        """
        This function creates a DroneController instance. The type of drone the
        DroneController pilots is determined by param which_type. The starting
        altitude is set to 0.1 as if a QGeoCoordinate is instantitated with 0.0
        (or null) for a starting value, the QGeoCoordinate is treated as a 2d
        coordinate instead of 3d. Can cause issues with reading altitudes later.

        :param name: str, name for DroneController
        :param lat: str, represents latitude for DroneController home location
        :param lng: str, represents longitude for DroneController home location
        :param which_type: str, represents the type of drone the DroneController mus
t pilot.
        """
        starting_point = QGeoCoordinate(float(lat), float(lng), 0.1)
        d = DroneController(name, starting_point, which_type)
        self.drone_status[name] = {}
        self.drones[name] = d


    @pyqtSlot(str, result=bool)
    def is_name_taken(self, name):
        """
        This function determines if the name the user has chosen for
        the drone has already been taken.

        :return a boolean expression
        """
        if name in self.drones.keys():
            return True
        return False



    @pyqtSlot(str, result=QGeoCoordinate)
    def get_drone_location(self, name):
        """
        This function returns a QGeoCoordinate containing
        the current location of DroneController with param name

        :param name: str, representing the name of the DroneController
        :return: QGeoCoordinate, representing the DroneControllers location
        """
```

```
        return self.drones[name].get_current_location()



    @pyqtSlot(str, result=QGeoCoordinate)
    def get_next_drone_location(self, name):
        """
        This function returns a QGeoCoordinate containing the location
        the DroneController with param name is travelling to.

        :param name: str, representing the name of the DroneController
        :return: QGeoCoordinate, representing the DroneControllers target destinatio
n
        """
        return self.drones[name].going_to()



    @pyqtSlot(str, str)
    def assign_flight_plan_to_drone(self, drone_name, fp_name):
        """
        This function assigns a FlightPlan with param fp_name
        to a DroneController with param drone_name. If the name is
        valid, if it hasn't already been assigned and if it's a new
        FlightPlan, it will be assigned to the DroneController.
        A path to the first coordinate in the FlightPlan will be
        plotted.

        :param drone_name: str, the name of the DroneController
        :param fp_name: str, the name of the FlightPlan
        """
        if fp_name in self.flight_plans.keys():
            if self.is_flight_plan_already_assigned(fp_name) == False:
                self.drones[drone_name].assign_flight_plan(fp_name)
            elif self.is_flight_plan_already_assigned(fp_name) == True:
                self.flightPlanAlreadyAssigned.emit(fp_name)
                return
        else:
            self.create_flight_plan(fp_name)
            self.drones[drone_name].assign_flight_plan(fp_name)
        self.plan_drone_path(drone_name)



    @pyqtSlot(str, result=bool)
    def is_flight_plan_already_assigned(self, fp_name):
        """
        This function returns true if the FlightPlan has been
        assigned to another DroneController already.

        :param fp_name: str, the name of the FlightPlan

        :return: a boolean expression
        """
        for drone in self.drones.values():
            if drone.get_flight_plan_name() == fp_name:
                return True
        return False



    @pyqtSlot(str)
    def begin_flight_plan(self, drone_name):
        """
        This function instructs the DroneController
        drone_name to begin its FlightPlan. First, the drone
        route is shown to the user with plot_drone_route()
        and the DroneController is instructed to takeoff().
        When the DroneController takes off, it is set as active
        will be picked up when the next request is made for drone
        information

        :param drone_name: str, the name of the DroneController
```

```
        """
        self.plot_drone_route(drone_name)
        self.tell_drone_to_takeoff(drone_name)


    @pyqtSlot(str, result=float)
    def get_drone_speed(self, drone_name):
        """
        This function returns the current speed of the
        DroneController with param drone_name

        :param drone_name: str, the name of the DroneController
        :return: float, the current speed
        """
        return self.drones[drone_name].get_current_speed()

    @pyqtSlot(str)
    def land_drone(self, drone_name):
        """
        This function lands the drone
        """
        self.drones[drone_name].land()


    @pyqtSlot(str)
    def run_test(self, drone_name):
        """
        This function is responbile for running the test
        to verify a working connection to the DATCS_Bebop drone

        :param drone_name: str
        """
        self.drones[drone_name].run_test()


    @pyqtSlot(result = QGeoCoordinate)
    def get_real_drone_location(self):
        """
        This function gets the current location of the
        DATCS_Bebop drone.
        """
        for drone in self.drones.values():
            if drone.which_type != "Sim":
                return drone.get_current_location()


    @pyqtSlot(str)
    def fork_vlc_and_view_drone_feed(self, drone_name):
        """
        This function creates a subprocess that opens VLC
        and displays the drone video stream using a .sdp file
        located in pyparrot.

        The path to the .sdp is found by creating another subprocess
        to run the find command in linux terminal
        """
        self.drones[drone_name].setup_video()
        out = subprocess.Popen(['find', '/home', '-name', 'bebop.sdp'],
                stdout= subprocess.PIPE, stderr=subprocess.STDOUT)
        stdout, stderr = out.communicate()
        output_list = str(stdout).split('\\n')
        filepath = output_list[0][2::] # first result is the one we want, splitting
removes 'b/'
        subprocess.Popen(["vlc", "file://" + str(filepath)])


    @pyqtSlot()
    def shutdown_feed_and_drone(self):
        """
        This function is responsible for shutting down the
        video feed and closing the connection to the DATCS_Bebop drone
        """
        for drone in self.drones.values():
```

```
            if drone.which_type != "Sim":
                drone.close_feed_and_disconnect()


    @pyqtSlot(str)
    def speedup(self, drone_name):
        """
        This function instructs the drone to speed up.
        As DroneController speed was used in calculating the
        path the drone must traverse, the path must be recalculated.

        :param drone_name: str
        """
        self.drones[drone_name].speedup()
        self.recalculate_path(drone_name)


    @pyqtSlot(str)
    def slowdown(self, drone_name):
        """
        This function instructs the drone to slow down.
        As DroneController speed was used in calculating the
        path the drone must traverse, the path must be recalculated.

        :param drone_name: str
        """
        self.drones[drone_name].slowdown()
        self.recalculate_path(drone_name)


    @pyqtSlot(str)
    def delete_drone(self, drone_name):
        """
        This function removes a DroneController with name
        param drone_name. The QML view is notified of this change

        :param drone_name: str
        """
        fp_name = self.drones[drone_name].get_flight_plan_name()
        if fp_name != '':
            self.flight_plans.pop(fp_name)
        self.drones.pop(drone_name)
        self.removeDronePath.emit(fp_name)


    @pyqtSlot(str)
    def call_drone_home(self, drone_name):
        """
        This function calls the DroneController with param drone_name
        back to its starting location. If the DroneController is piloting
        a SimulatedDrone, the DroneController is marked as inactive,
        the FlightPlan is reset, and the drone begins moving home.
        This information is displayed to the user.

        :param drone_name: str, DroneController name
        """
        if self.drones[drone_name].which_type == "Sim":
            self.drones[drone_name].is_active = False
            self.reset_drone_flight_plan(drone_name)
            self.drones[drone_name].reset_time_spent_travelling()
            self.drones[drone_name].arrived_at_start_point = False
            home = self.drones[drone_name].get_home()
            self.plan_drone_path(drone_name, home)
            self.drones[drone_name].moving_home = True
            self.movingHome.emit(drone_name)
        else:
            self.drones[drone_name].return_home()
            self.movvingHome.emit(drone_name)

################################################################################

    def issue_stop_message(self, drone_names):
```

```
        """
        This function stops all drones in param
        drone_names from flying. The user console
        will display the drones that are stopping.

        :param drone_names: list, string list of drone_names
        """
        for name in drone_names:
            self.drones[name].stop()
            update_string = "Stopping Drone " + name
            self.updateATCConsole.emit(update_string)
            self.stopDrone.emit(name)


    def revive_stopped_drones(self):
        """
        This function revives all stopped drones.
        """
        for name in list(self.drones.keys()):
            self.drones[name].resume()


    def request_drone_info(self):
        """
        This function polls each active and moving drone and moves them along
        their respective FlightPlans or paths home. Once all have moved,
        their information gets processed and all drones are examined for a
        potential collision.
        """
        for k in list(self.drones.keys()):
            if self.drones[k].is_drone_active() == True and self.drones[k].is_drone_
waiting() == False:
                #self.sendDroneInfo.emit(k)
                if self.drones[k].which_type == "Sim":
                    self.tell_drone_to_move_to(k)
                else:
                    self.move_real_drone_to(k)
            if self.drones[k].is_drone_homeward_bound() == True:
                #self.sendDroneInfo.emit(k)
                if self.drones[k].which_type == "Sim":
                    self.tell_drone_to_move_home(k)
                else:
                    self.call_real_drone_home(k)
        self.process_information()


    def process_information(self):
        """
        After requesting information from each drone, the CollisionPredictor
        will examine each drones information to determine if there is a possibilty
        of a collision. If there is, the drones in danger will be stopped.
        """
        self.cp.sitrep(self.drones.values())
        self.cp.decide()
        drones_to_be_stopped = self.cp.stop_which_drones()
        if len(drones_to_be_stopped) == 0:
            self.revive_stopped_drones()
        else:
            self.issue_stop_message(drones_to_be_stopped)


    def tell_drone_to_takeoff(self, drone_name):
        """
        This function instructs the DroneController with
        param name to takeoff

        :param drone_name: str, the name of the DroneController
        """
        self.drones[drone_name].takeoff()


    def tell_drone_to_stop(self, drone_name):
        """
```

```
        This function instructs the DroneController with
        param name to stop

        :param drone_name: str, the name of the DroneController
        """
        self.drones[drone_name].stop()


    def tell_drone_to_land(self, drone_name):
        """
        This function instructs the DroneController with
        param name to land

        :param drone_name: str, the name of the DroneController
        """
        self.drones[drone_name].land()


    def tell_drone_to_return_home(self, drone_name):
        """
        This function instructs the DroneController with
        param name to return home

        :param drone_name: str, the name of the DroneController
        """
        self.drones[drone_name].return_home()


    def tell_drone_to_resume(self, drone_name):
        """
        This function instructs the DroneController with
        param name to resume

        :param drone_name: str, the name of the DroneController
        """
        self.drones[drone_name].resume()


    def get_drone_air_time(self, drone_name):
        """
        This function returns the time a DroneController has spent travelling
        to the next location in its FlightPlan. This time is used to index the
        DroneControllers place in the proposed path between coordinates.

        :param drone_name: str, the name of the DroneController
        """
        index = self.drones[drone_name].get_time_spent_travelling()
        index = int(index)
        return index


    def has_drone_reached_the_starting_point(self, drone_name):
        """
        This function evaluates whether the DroneController has moved past
        the starting point in the FlightPlan

        :param drone_name: str, the name of the DroneController

        :return: a boolean expression
        """
        return self.drones[drone_name].arrived_at_start_point


    def update_console(self, drone_name, fp_name):
        """
        This function updates the console in the user view.

        :param drone_name: str, the name of the DroneController
        :param fp_name: str, the name of the FlightPlan
        """
        update_string = "Drone " + drone_name + " arrived at " + fp_name + ", Coordi
nate: "
        coord = self.drones[drone_name].get_current_location()
```

```
        update_string += coord.toString(QGeoCoordinate.Degrees)
        self.updateATCConsole.emit(update_string)


    def tell_drone_to_move_to(self, drone_name):
        """
        This function instructs to move the DroneController with param
        drone_name. First, we get the index of the coordinates we must
        traverse between coordinates. If we have traversed all points
        between points, then we need to check if the FlightPlan has been
        completed. After the DroneController has passed each FlightPlan
        coordinate, a new path to the next coordinate is planned. The
        DroneController will be ordered to move to the next coordinate in
        this newly created path on the next timer iteration.

        :param drone_name: str, the name of the DroneController
        """
        # get the index of the coordinates we must traverse between flight plan coor
dinates
        index = self.get_drone_air_time(drone_name)

        # determine if the drone has arrived at it's starting point
        past_the_start = self.has_drone_reached_the_starting_point(drone_name)

        fp_name = self.drones[drone_name].get_flight_plan_name()

        # if we have traveresed all points between points...
        if index == len(self.drone_paths[drone_name]):
            inx = self.flight_plans[fp_name].get_current_index()

            # are we there yet?
            if inx == self.flight_plans[fp_name].get_flight_plan_length() - 1:
                self.drones[drone_name].is_active = False
                self.drones[drone_name].reset_time_spent_travelling()
                self.update_console(drone_name, fp_name)
                self.flight_plans[fp_name].reset_flight_plan()
                self.flightPlanFinished.emit(drone_name)
                return

            # have we arrived at the start point, or a flight plan point?
            if past_the_start == False:
                #First time round, no incrementing fp index
                self.drones[drone_name].arrived_at_start_point = True
            # otherwise, we can move on
            elif inx >= 0 and past_the_start == True:
                self.flight_plans[fp_name].increment_index()
                # inform the user that we have reached the location
                self.update_console(drone_name, fp_name)

            # reset the time spent travelling between points
            self.drones[drone_name].reset_time_spent_travelling()
            new_index = self.get_drone_air_time(drone_name)

            # plot the points we must traverse
            self.plan_drone_path(drone_name)
            # get the next point we need
            next_point = self.drone_paths[drone_name][0] #index has been reset at th
is point
            # instruct the drone to move
            self.drones[drone_name].go_here(next_point)
            self.moveDrone.emit(drone_name, next_point)

        else:
            # we travel towards the next point
            next_point = self.drone_paths[drone_name][index]
            self.drones[drone_name].go_here(next_point)
            self.moveDrone.emit(drone_name, self.get_drone_location(drone_name)) #,
next_point)


    def get_distance_to_next_point(self, current_location, next_point):
        """
        This function calculates the distance between param
```

```
        current_location and param next_point in meters.

        :param current_location: QGeoCoordinate
        :param next_point: QGeoCoordinate

        :return distance: float, represents the distance in meters
        """
        distance = current_location.distanceTo(next_point)
        return distance


    def calculate_time_to_arrival(self, distance, speed):
        """
        This function calculates the time it will take to arrive
        at a location given the distance and speed. The time is returned
        in seconds.

        :param distance: the distance in meters
        :param speed: the current speed in kmph

        :return time: float, in seconds
        """
        time = distance / (speed * 0.277778) ##convert kmph to m/s
        return time


    def plan_drone_path(self, drone_name, home = None):
        """
        This function plans the DroneControllers path between
        coordinates and stores it in the drone_paths dictionary
        """
        self.drone_paths[drone_name] = self.plot_path(drone_name, home)


    def plot_path(self, drone_name, home = None):
        """
        This function accepts string param drone_name, and using drone_name,
        identifies the points the drone must traverse on its path to its
        next location. The distance between points in meters and current drone speed
        in kilometers per hour is needed to calculate the duration of the drone's fl
ight
        to the next point. For each second given, a point on the line in stored in
        list path, which is stored by the AirTrafficController. The AirTrafficContro
ller
        will move the drone along this path to the next point unless something occur
s.

        :param drone_name: string representing the drones name

        :returns a list of QGeoCoordinates representing a point on the line between
            drone_start_point and drone_next_point
        """
        drone_start_point = self.drones[drone_name].get_current_location()
        drone_flight_plan = self.drones[drone_name].get_flight_plan_name()

        if home != None:
            drone_next_point = home
            self.drones[drone_name].point_moving_towards = drone_next_point
        else:
            drone_next_point = self.flight_plans[drone_flight_plan].get_coordinates(
)
            self.drones[drone_name].point_moving_towards = drone_next_point

        distance = self.get_distance_to_next_point(drone_start_point, drone_next_poi
nt)
        speed = self.get_drone_speed(drone_name)
        ETA = self.calculate_time_to_arrival(distance, speed)
        ETA = int(ETA) #integer representation needed for following calculations
        if (ETA <= 0):
            ETA = 1.0
        i = ETA
        lat, lng = 0.0, 0.0
        chk = False
```

```
        path = []
        while chk != True:
            lat = drone_start_point.latitude() * i / ETA + drone_next_point.latitude
() * (ETA - i) / ETA
            lng = drone_start_point.longitude() * i / ETA + drone_next_point.longitu
de() * (ETA - i) / ETA
            i -= 1
            path.append(QGeoCoordinate(lat, lng, drone_next_point.altitude()))
            if i == 0:
                chk = True
        path.append(drone_next_point) # add the point to travel to at the end for co
mpleteness
        return path


    def plot_drone_route(self, drone_name):
        """
        This function displays the assigned flight plan to the
        user in QML view

        :param drone_name: str, DroneController name
        """
        fp_name = self.drones[drone_name].get_flight_plan_name()
        full_fp_coordinates = self.flight_plans[fp_name].get_entire_plan()
        geocoord_list = []
        for row in full_fp_coordinates:
            lat = row['lat']
            lng = row['lng']
            coord = QGeoCoordinate(lat, lng)
            geocoord_list.append(coord)
        self.plotDroneRouteViz.emit(geocoord_list, fp_name)


    def reset_drone_flight_plan(self, drone_name):
        """
        This function resets the flight plan assigned to
        DroneController with param drone_name. If the flight
        plan name is '', the flight plan hasn't been assigned
        and the drone is at its home location.

        :param drone_name: str
        """
        fp = self.drones[drone_name].get_flight_plan_name()
        if fp != "": # at home
            self.flight_plans[fp].reset_flight_plan()


    def tell_drone_to_move_home(self, drone_name):
        """
        This function instructs the DroneController with param
        drone_name to move towards its home location. Similarly to
        how a path is plotted between flight plan coordinates,
        a path is plotted between the drones current location
        and the home location. The DroneController will continue
        traversing this path until it has arrived at its destination.

        :param drone_name: str
        """
        index = self.drones[drone_name].get_time_spent_travelling_towards_home()
        if index == len(self.drone_paths[drone_name]):
            self.drones[drone_name].moving_home = False
            home_string = "Drone " + drone_name + " has arrived home"
            self.arrivedHome.emit(home_string, drone_name)
            return
        next_point = self.drone_paths[drone_name][index]
        self.drones[drone_name].go_here(next_point)
        self.moveDrone.emit(drone_name, self.get_drone_location(drone_name))
        self.drones[drone_name].time_spent_travelling_towards_home += 1


    def recalculate_path(self, drone_name):
        """
        This function recalculates the path between
```

```
        the DroneController with param drone_name.

        :param drone_name: str
        """
        # reset the time spent travelling between points
        self.drones[drone_name].reset_time_spent_travelling()
        new_index = self.get_drone_air_time(drone_name)
        # plot the points we must traverse
        self.plan_drone_path(drone_name)
        # get the next point we need
        next_point = self.drone_paths[drone_name][0] #index has been reset at this p
oint
        # instruct the drone to move
        self.drones[drone_name].go_here(next_point)
        self.moveDrone.emit(drone_name, next_point)


    def move_real_drone_to(self, name):
        """
        This function is responsible for moving the DATCS_Bebop
        drone to a fixed geographic location.
        """
        if self.drones[name].ready_to_move_again():
            fp_name = self.drones[name].get_flight_plan_name()
            if self.flight_plans[fp_name].is_flight_plan_complete() == False:
                next_point = self.flight_plans[fp_name].get_coordinates()
                self.drones[drone_name].go_here(next_point)
                self.moveDrone.emit(drone_name, next_point)
                self.flight_plans[fp_name].increment_index()
```

```
# Author: Ronan Donohue
# Number: c00208501

__license__ = "MIT"
__revision__ = "CollisionPredictor.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'

import itertools
from math import pi, cos, radians
from PyQt5.QtPositioning import QGeoCoordinate

class CollisionPredictor():
    """
    This is a class invoked by an AirTrafficController object to
    determine if any collisions are about to occur.

    :param centerof_map: a QGeoCoordinate object. The coordinate latitude
    is needed in some calculations required by the CollisionPredictor.
    """
    def __init__(self, center_of_map):
        """
        Initializes the CollisionPredictor object.
        """
        self.active_drones_status = {}
        self.radius = 6371000 # radius of Earth in meters
        self.map_lat = center_of_map.latitude()
        self.cos_map_lat = cos(radians(self.map_lat))
        self.acceptable_alt_difference = 2.0 #meters
        self.acceptable_distance = 30.0 #meters


    def sitrep(self, current_situation):
        """
        Situational Report. Param current_situation, type list, is passed in
        containing all the current airborne drones from AirTrafficController.
        Sets the status of all active drones to TBD.
        """
        self.current_situation = current_situation
        for drone in current_situation:
            self.active_drones_status[drone.name] = "TBD"


    def decide(self):
        """
        Calls determine() on each drone with every other drone
        """
        for a, b in itertools.combinations(self.current_situation, 2):
            self.determine(a, b)


    def on_line(self, line, point):
        """
        Determines if param point lies on param line

        :param line: a list containing 2 points
        :param point: a tuple, consisting of two points in x,y space

        :return: A boolean expression
        """
        if (
            (point[0] <= max(line[0][0], line[1][0])
                and point[0] <= min(line[0][0], line[1][0]))
            and
            (point[1] <= max(line[0][1], line[1][1])
                and point[0] <= min(line[0][1], line[1][1]))
            ):
            return True
        return False


    def direction(self, a, b, c):
        """
        Get the direction between a, b, and c
```

```
        :param a: a tuple, consisting of coordinates in x, y space
        :param b: a tuple, consisting of coordinates in x, y space
        :param c: a tuple, consisting of coordinates in x, y space

        :return: an integer between 0 and 2.
            0: means the points are colinear
            1: means the points are clockwise
            2: means the points are anti-clockwise
        """
        val = (b[1] - a[1]) * (c[0] - b[0]) - (b[0] - a[0]) * (c[1] - b[1])
        if val == 0:
            return 0 # colinear
        elif val < 0:
            return 2 # anti-clockwise
        return 1 # clockwise


    def intersection(self, lineA, lineB):
        """
        Determines if param lineA intersects param lineB

        :param lineA: a list, containing two points
        :param lineB: a list, containing two points

        :return: a boolean expression
        """
        dirA = self.direction(lineA[0], lineA[1], lineB[0])
        dirB = self.direction(lineA[0], lineA[1], lineB[1])
        dirC = self.direction(lineB[0], lineB[1], lineA[0])
        dirD = self.direction(lineB[0], lineB[1], lineA[1])

        if dirA != dirB and dirC != dirD:
            return True

        # when x of lineB is on lineA
        if dirA == 0 and self.on_line(lineA, lineB[0]):
            return True
        # when y of lineB is on lineA
        if dirB == 0 and self.on_line(lineA, lineB[1]):
            return True
        # when x of lineA is on lineB
        if dirC == 0 and self.on_line(lineB, lineA[0]):
            return True
        # when y of lineA is on lineB
        if dirD == 0 and self.on_line(lineB, lineA[1]):
            return True

        return False


    def to_xy(self, point):
        """
        The function uses a formula that is a simple approximation of
        map projections like transverse Mercator or UTM
        where ¿ = longitude and ¿ = latitutde. ¿0 represents the
        latitude of the center point for the map.

        Does not account for the curvature of the earth.

        :Formula
        x = r ¿ cos(¿0)
        y = r ¿

        :param point: a QGeoCoordinate, either the current location
        of a Drone or the point the Drone is journeying towards

        :return: lat and lng converted to 2d x and y values
        :rtype: a tuple, consisting of two floats
        """
        lam = point.latitude()
        phi = point.longitude()
        return (self.radius * radians(lam) * self.cos_map_lat,
```

```
                self.radius * radians(phi))

    def determine(self, droneA, droneB):
        """
        This function determines if a collision is imminent between
        two param DroneController objects.
        If the two drones are at the same altitude, their paths intersect
        and the distance between them is less than self.acceptable_distance,
        one must stop and wait.

        :param droneA: a DroneController object
        :param droneB: a DroneCOntroller object
        """

        # First, get altitudes
        altA = droneA.get_current_location().altitude()
        altB = droneB.get_current_location().altitude()

        drop = False

        if abs(altA-altB) <= self.acceptable_alt_difference:
            # Drones are at the same altitude
            # convert lat/lng to x/y format for all points
            # create two line segments, representing drone paths
            # determine if points intersect
            # if points intersect, collision imminent
            droneA_current_point = droneA.get_current_location()
            droneA_next_point = droneA.get_point_travelling_to()

            droneB_current_point = droneB.get_current_location()
            droneB_next_point = droneB.get_point_travelling_to()

            pointAXY = self.to_xy(droneA_current_point)
            pointBXY = self.to_xy(droneA_next_point)
            pointCXY = self.to_xy(droneB_current_point)
            pointDXY = self.to_xy(droneB_next_point)

            lineA = [pointAXY, pointBXY]
            lineB = [pointCXY, pointDXY]

            if self.intersection(lineA, lineB):
                # So our altitudes are similar, and our lines intersect
                # It is a matter of time...
                # But are we near one another? And are we heading towards one anothe
r?

                distance = droneA_current_point.distanceTo(droneB_current_point)

                # bearing = droneA_current_point.azimuthTo(droneB_current_point)
                # print("Bearing:", bearing)

                #if int(bearing) in (180, 181, 182, 179, 178, 360, 359, 358, 0, 1, 2
):
                #    print("Head on")
                #    drop = True

                if distance <= self.acceptable_distance:
                    # The drones are now too close together
                    # Time to act!
                    self.update(droneA, droneB, drop)


    def update(self, droneA, droneB, drop = False):
        """
        The drone that has been travelling the shortest amount of time must
        wait. This accounts for drones that are travelling to home also. If a
        head-on collision seems likely, one of the drones must adjust their
        altitude.

        :param droneA: a DroneController object
        :param droneB: a DroneController object
        """
```

```
        if ((droneA.time_spent_travelling <= droneB.time_spent_travelling
        or droneA.time_spent_travelling_towards_home <= droneB.time_spent_travelling
)
        or (droneA.time_spent_travelling <= droneB.time_spent_travelling_towards_hom
e
        or droneA.time_spent_travelling_towards_home <= droneB.time_spent_travelling
_towards_home)):
            if drop:
                print("DroneA alt before:", droneA.altitude_reading)
                droneA.lower_altitude(2)
                print("DroneA alt after:", droneA.altitude_reading)
            self.active_drones_status[droneA.name] = "Waiting"
        else:
            if drop:
                print("DroneA alt before:", droneB.altitude_reading)
                droneB.lower_altitude(2)
                print("DroneB alt before:", droneB.altitude_reading)
            self.active_drones_status[droneB.name] = "Waiting"


    def stop_which_drones(self):
        """
        The drones whos status has been set to waiting are passed
        back to the AirTrafficController to be stopped.

        :return names: a list of drones that need to wait.
        """
        names = []
        for k, v in self.active_drones_status.items():
            if v == "Waiting":
                names.append(k)
        return names
```

```python
# Author: Ronan Donohue
# Number: c00208501

from PyQt5.QtGui import QGuiApplication
from PyQt5.QtQml import QQmlApplicationEngine, qmlRegisterType
from PyQt5.QtCore import QObject, QUrl, pyqtSignal, pyqtProperty
from PyQt5.QtPositioning import QGeoCoordinate

from AirTrafficController import AirTrafficController

if __name__ == "__main__":
    import os
    import sys

    app = QGuiApplication(sys.argv)

    engine = QQmlApplicationEngine()

    atc = AirTrafficController()

    engine.rootContext().setContextProperty("atc", atc)

    engine.load(QUrl.fromLocalFile("datcs_main.qml"))

    if not engine.rootObjects():
        sys.exit(-1)

    engine.quit.connect(app.quit)
    sys.exit(app.exec_())
```

```
"""
Bebop class holds all of the methods needed to pilot the drone from python and to as
k for sensor
data back from the drone

Author: Amy McGovern, dramymcgovern@gmail.com
"""
import time
from pyparrot.networking.wifiConnection import WifiConnection
from pyparrot.utils.colorPrint import color_print
from pyparrot.commandsandsensors.DroneCommandParser import DroneCommandParser
from pyparrot.commandsandsensors.DroneSensorParser import DroneSensorParser
from datetime import datetime

class BebopSensors:
    def __init__(self):
        self.sensors_dict = dict()
        self.RelativeMoveEnded = False
        self.CameraMoveEnded_tilt = False
        self.CameraMoveEnded_pan = False
        self.flying_state = "unknown"
        self.flat_trim_changed = False
        self.max_altitude_changed = False
        self.max_distance_changed = False
        self.no_fly_over_max_distance = False
        self.max_tilt_changed = False
        self.max_pitch_roll_rotation_speed_changed = False
        self.max_vertical_speed = False
        self.max_rotation_speed = False
        self.hull_protection_changed = False
        self.outdoor_mode_changed = False
        self.picture_format_changed = False
        self.auto_white_balance_changed = False
        self.exposition_changed = False
        self.saturation_changed = False
        self.timelapse_changed = False
        self.video_stabilization_changed = False
        self.video_recording_changed = False
        self.video_framerate_changed = False
        self.video_resolutions_changed = False

        # default to full battery
        self.battery = 100

        # this is optionally set elsewhere
        self.user_callback_function = None


    def set_user_callback_function(self, function, args):
        """
        Sets the user callback function (called everytime the sensors are updated)

        :param function: name of the user callback function
        :param args: arguments (tuple) to the function
        :return:
        """
        self.user_callback_function = function
        self.user_callback_function_args = args


    def update(self, sensor_name, sensor_value, sensor_enum):
        if (sensor_name is None):
            print("Error empty sensor")
            return


        if (sensor_name, "enum") in sensor_enum:
            # grab the string value
            if (sensor_value is None or sensor_value > len(sensor_enum[(sensor_name,
 "enum")])):
                value = "UNKNOWN_ENUM_VALUE"
            else:
                enum_value = sensor_enum[(sensor_name, "enum")][sensor_value]
```

```
                value = enum_value

            self.sensors_dict[sensor_name] = value

        else:
            # regular sensor
            self.sensors_dict[sensor_name] = sensor_value

        # some sensors are saved outside the dictionary for internal use (they are a
lso in the dictionary)
        if (sensor_name == "FlyingStateChanged_state"):
            self.flying_state = self.sensors_dict["FlyingStateChanged_state"]

        if (sensor_name == "moveToChanged_status"):
            self.move_status = self.sensors_dict["moveToChanged_status"]

        if (sensor_name == "PilotingState_FlatTrimChanged"):
            self.flat_trim_changed = True

        if (sensor_name == "moveByEnd_dX"):
            self.RelativeMoveEnded = True

        if (sensor_name == "OrientationV2_tilt"):
            self.CameraMoveEnded_tilt = True

        if (sensor_name == "OrientationV2_pan"):
            self.CameraMoveEnded_pan = True

        if (sensor_name == "MaxAltitudeChanged_current"):
            self.max_altitude_changed = True

        if (sensor_name == "MaxDistanceChanged_current"):
            self.max_distance_changed = True

        if (sensor_name == "NoFlyOverMaxDistanceChanged_shouldNotFlyOver"):
            self.no_fly_over_max_distance_changed = True

        if (sensor_name == "MaxTiltChanged_current"):
            self.max_tilt_changed = True

        if (sensor_name == "MaxPitchRollRotationSpeedChanged_current"):
            self.max_pitch_roll_rotation_speed_changed = True

        if (sensor_name == "MaxVerticalSpeedChanged_current"):
            self.max_vertical_speed_changed = True

        if (sensor_name == "MaxRotationSpeedChanged_current"):
            self.max_rotation_speed_changed = True

        if (sensor_name == "HullProtectionChanged_present"):
            self.hull_protection_changed = True

        if (sensor_name == "OutdoorChanged_present"):
            self.outdoor_mode_changed = True

        if (sensor_name == "BatteryStateChanged_battery_percent"):
            self.battery = sensor_value

        if (sensor_name == "PictureFormatChanged_type"):
            self.picture_format_changed = True

        if (sensor_name == "AutoWhiteBalanceChanged_type"):
            self.auto_white_balance_changed = True

        if (sensor_name == "ExpositionChanged_value"):
            self.exposition_changed = True

        if (sensor_name == "SaturationChanged_value"):
            self.saturation_changed = True

        if (sensor_name == "TimelapseChanged_enabled"):
            self.timelapse_changed = True
```

```
        if (sensor_name == "VideoStabilizationModeChanged_mode"):
            self.video_stabilization_changed = True

        if (sensor_name == "VideoRecordingModeChanged_mode"):
            self.video_recording_changed = True

        if (sensor_name == "VideoFramerateChanged_framerate"):
            self.video_framerate_changed = True

        if (sensor_name == "VideoResolutionsChanged_type"):
            self.video_resolutions_changed = True

        # call the user callback if it isn't None
        if (self.user_callback_function is not None):
            self.user_callback_function(self.user_callback_function_args)


    def __str__(self):
        str = "Bebop sensors: %s" % self.sensors_dict
        return str


class DATCS_Bebop():

    def __init__(self, drone_type="Bebop2"):
        """
        Create a new Bebop object.  Assumes you have connected to the Bebop's wifi

        """
        self.drone_type = drone_type

        self.drone_connection = WifiConnection(self, drone_type=drone_type)

        # intialize the command parser
        self.command_parser = DroneCommandParser()

        # initialize the sensors and the parser
        self.sensors = BebopSensors()
        self.sensor_parser = DroneSensorParser(drone_type=drone_type)


    def set_user_sensor_callback(self, function, args):
        """
        Set the (optional) user callback function for sensors.  Every time a sensor
        is updated, it calls this function.

        :param function: name of the function
        :param args: tuple of arguments to the function
        :return: nothing
        """
        self.sensors.set_user_callback_function(function, args)


    def update_sensors(self, data_type, buffer_id, sequence_number, raw_data, ack):
        """
        Update the sensors (called via the wifi or ble connection)

        :param data: raw data packet that needs to be parsed
        :param ack: True if this packet needs to be ack'd and False otherwise
        """
        #print("data type is %d buffer id is %d sequence number is %d " % (data_type
, buffer_id, sequence_number))
        sensor_list = self.sensor_parser.extract_sensor_values(raw_data)
        #print(sensor_list)
        if (sensor_list is not None):
            for sensor in sensor_list:
                (sensor_name, sensor_value, sensor_enum, header_tuple) = sensor
                if (sensor_name is not None):
                    self.sensors.update(sensor_name, sensor_value, sensor_enum)
                else:
                    color_print("data type %d buffer id %d sequence number %d" % (da
ta_type, buffer_id, sequence_number), "WARN")
                    color_print("This sensor is missing (likely because we don't nee
```

```
d it)", "WARN")

        if (ack):
            self.drone_connection.ack_packet(buffer_id, sequence_number)


    def connect(self, num_retries):
        """
        Connects to the drone and re-tries in case of failure the specified number o
f times.  Seamlessly
        connects to either wifi or BLE depending on how you initialized it

        :param: num_retries is the number of times to retry

        :return: True if it succeeds and False otherwise
        """

        # special case for when the user tries to do BLE when it isn't available
        if (self.drone_connection is None):
            return False

        connected = self.drone_connection.connect(num_retries)
        return connected


    def disconnect(self):
        """
        Disconnect the BLE connection.  Always call this at the end of your programs
 to
        cleanly disconnect.

        :return: void
        """
        self.drone_connection.disconnect()


    def ask_for_state_update(self):
        """
        Ask for a full state update (likely this should never be used but it can be
called if you want to see
        everything the bebop is storing)

        :return: nothing but it will eventually fill the sensors with all of the sta
te variables as they arrive
        """
        command_tuple = self.command_parser.get_command_tuple("common", "Common", "A
llStates")
        return self.drone_connection.send_noparam_command_packet_ack(command_tuple)


    def flat_trim(self, duration=0):
        """
        Sends the flat_trim command to the bebop. Gets the codes for it from the xml
 files.
        :param duration: if duration is greater than 0, waits for the trim command t
o be finished or duration to be reached
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "FlatTrim")
        self.drone_connection.send_noparam_command_packet_ack(command_tuple)

        if (duration > 0):
            # wait for the specified duration
            start_time = datetime.now()
            new_time = datetime.now()
            diff = (new_time - start_time).seconds + ((new_time - start_time).micros
econds / 1000000.0)

            while (not self.sensors.flat_trim_changed and diff < duration):
                self.smart_sleep(0.1)

                new_time = datetime.now()
                diff = (new_time - start_time).seconds + ((new_time - start_time).mi
```

```
croseconds / 1000000.0)


    def takeoff(self):
        """
        Sends the takeoff command to the bebop.  Gets the codes for it from the xml
files.  Ensures the
        packet was received or sends it again up to a maximum number of times.

        :return: True if the command was sent and False otherwise
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "TakeOff")
        self.drone_connection.send_noparam_command_packet_ack(command_tuple)


    def safe_takeoff(self, timeout):
        """
        Sends commands to takeoff until the Bebop reports it is taking off

        :param timeout: quit trying to takeoff if it takes more than timeout seconds
        """

        start_time = time.time()
        # take off until it really listens
        while (self.sensors.flying_state != "takingoff" and (time.time() - start_tim
e < timeout)):
            if (self.sensors.flying_state == "emergency"):
                return
            success = self.takeoff()
            self.smart_sleep(1)

        # now wait until it finishes takeoff before returning
        while ((self.sensors.flying_state not in ("flying", "hovering") and
                    (time.time() - start_time < timeout))):
            if (self.sensors.flying_state == "emergency"):
                return
            self.smart_sleep(1)


    def land(self):
        """
        Sends the land command to the bebop.  Gets the codes for it from the xml fil
es.  Ensures the
        packet was received or sends it again up to a maximum number of times.

        :return: True if the command was sent and False otherwise
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "Landing")
        return self.drone_connection.send_noparam_command_packet_ack(command_tuple)


    def emergency_land(self):
        """
        Sends the land command to the bebop on the high priority/emergency channel.
        Gets the codes for it from the xml files.  Ensures the
        packet was received or sends it again up to a maximum number of times.

        :return: True if the command was sent and False otherwise
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "Landing")
        return self.drone_connection.send_noparam_high_priority_command_packet(comma
nd_tuple)


    def is_landed(self):
        """
        Returns true if it is landed or emergency and False otherwise
        :return:
        """
        if (self.sensors.flying_state in ("landed", "emergency")):
```

```
            return True
        else:
            return False


    def safe_land(self, timeout):
        """
        Ensure the Bebop lands by sending the command until it shows landed on senso
rs
        """
        start_time = time.time()

        while (self.sensors.flying_state not in ("landing", "landed") and (time.time
() - start_time < timeout)):
            if (self.sensors.flying_state == "emergency"):
                return
            color_print("trying to land", "INFO")
            success = self.land()
            self.smart_sleep(1)

        while (self.sensors.flying_state != "landed" and (time.time() - start_time <
 timeout)):
            if (self.sensors.flying_state == "emergency"):
                return
            self.smart_sleep(1)


    def smart_sleep(self, timeout):
        """
        Don't call time.sleep directly as it will mess up BLE and miss WIFI packets!
  Use this
        which handles packets received while sleeping

        :param timeout: number of seconds to sleep
        """
        self.drone_connection.smart_sleep(timeout)


    def _ensure_fly_command_in_range(self, value):
        """
        Ensure the fly direct commands are in range

        :param value: the value sent by the user
        :return: a value in the range -100 to 100
        """
        if (value < -100):
            return -100
        elif (value > 100):
            return 100
        else:
            return value


    def fly_direct(self, roll, pitch, yaw, vertical_movement, duration):
        """
        Direct fly commands using PCMD.  Each argument ranges from -100 to 100.  Num
bers outside that are clipped
        to that range.

        Note that the xml refers to gaz, which is apparently french for vertical mov
ements:
        http://forum.developer.parrot.com/t/terminology-of-gaz/3146

        :param roll:
        :param pitch:
        :param yaw:
        :param vertical_movement:
        :return:
        """

        my_roll = self._ensure_fly_command_in_range(roll)
        my_pitch = self._ensure_fly_command_in_range(pitch)
        my_yaw = self._ensure_fly_command_in_range(yaw)
```

```
        my_vertical = self._ensure_fly_command_in_range(vertical_movement)

        # print("roll is %d pitch is %d yaw is %d vertical is %d" % (my_roll, my_pit
ch, my_yaw, my_vertical))
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "PCMD")

        self.drone_connection.send_pcmd_command(command_tuple, my_roll, my_pitch, my
_yaw, my_vertical, duration)


    def flip(self, direction):
        """
        Sends the flip command to the bebop.  Gets the codes for it from the xml fil
es. Ensures the
        packet was received or sends it again up to a maximum number of times.
        Valid directions to flip are: front, back, right, left

        :return: True if the command was sent and False otherwise
        """
        fixed_direction = direction.lower()
        if (fixed_direction not in ("front", "back", "right", "left")):
            print("Error: %s is not a valid direction.  Must be one of %s" % directi
on, "front, back, right, or left")
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3",

  "Animations", "Flip", fixed_direction)
        # print command_tuple
        # print enum_tuple

        return self.drone_connection.send_enum_command_packet_ack(command_tuple, enu
m_tuple)


    def move_relative(self, dx, dy, dz, dradians):
        """
        Move relative to our current position and pause until the command is done.
Note that
        EVERY time we tested flying relative up (e.g. negative z) it did additional
lateral moves
        that were unnecessary.  I'll be posting this to the development board but, u
ntil then,
        I recommend only using dx, dy, and dradians which all seem to work well.

        :param dx: change in front axis (meters)
        :param dy: change in right/left (positive is right) (meters)
        :param dz: change in height (positive is DOWN) (meters)
        :param dradians: change in heading in radians

        :return: nothing
        """

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "moveBy")
        param_tuple = [dx, dy, dz, dradians]  # Enable
        param_type_tuple = ['float', 'float', 'float', 'float']
        #reset the bit that tells when the move ends
        self.sensors.RelativeMoveEnded = False

        # send the command
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple,
param_type_tuple)

        # sleep until it ends
        while (not self.sensors.RelativeMoveEnded):
            self.smart_sleep(0.01)


    def start_video_stream(self):
```

```
        """
        Sends the start stream command to the bebop. The bebop will start streaming
        RTP packets on the port defined in wifiConnection.py (55004 by default).
        The packets can be picked up by opening an approriate SDP file in a media
        player such as VLC, MPlayer, FFMPEG or OpenCV.

        :return: nothing
        """

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "MediaStre
aming", "VideoEnable")
        param_tuple = [1] # Enable
        param_type_tuple = ['u8']
        self.drone_connection.send_param_command_packet(command_tuple,param_tuple,pa
ram_type_tuple)


    def stop_video_stream(self):
        """
        Sends the stop stream command to the bebop. The bebop will stop streaming
        RTP packets.

        :return: nothing
        """

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "MediaStre
aming", "VideoEnable")
        param_tuple = [0] # Disable
        param_type_tuple = ['u8']
        self.drone_connection.send_param_command_packet(command_tuple,param_tuple,pa
ram_type_tuple)


    def set_video_stream_mode(self,mode='low_latency'):
        """
        Set the video mode for the RTP stream.
        :param: mode: one of 'low_latency', 'high_reliability' or 'high_reliability_
low_framerate'

        :return: True if the command was sent and False otherwise
        """

        # handle case issues
        fixed_mode = mode.lower()

        if (fixed_mode not in ("low_latency", "high_reliability", "high_reliability_
low_framerate")):
            print("Error: %s is not a valid stream mode.  Must be one of %s" % (mode
, "low_latency, high_reliability or high_reliability_low_framerate"))
            print("Ignoring command and returning")
            return False


        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3",

  "MediaStreaming", "VideoStreamMode", mode)

        return self.drone_connection.send_enum_command_packet_ack(command_tuple,enum
_tuple)


    def pan_tilt_camera(self, tilt_degrees, pan_degrees):
        """
        Send the command to pan/tilt the camera by the specified number of degrees i
n pan/tilt

        Note, this only seems to work in small increments.  Use pan_tilt_velocity to
 get the camera to look
        straight downward

        :param tilt_degrees: tilt degrees
        :param pan_degrees: pan degrees
```

```
        :return:
        """
        if(self.drone_type == "Bebop2"):
            command_tuple = self.command_parser.get_command_tuple("ardrone3", "Camer
a", "OrientationV2")

            self.drone_connection.send_param_command_packet(command_tuple, param_tup
le=[tilt_degrees, pan_degrees],
                                                                param_type_tuple=['float
', 'float'], ack=False)
        else:
            command_tuple = self.command_parser.get_command_tuple("ardrone3", "Camer
a", "Orientation")

            self.drone_connection.send_param_command_packet(command_tuple, param_tup
le=[tilt_degrees, pan_degrees],
                                                                param_type_tuple=['i8',
'i8'], ack=False)


    def pan_tilt_camera_velocity(self, tilt_velocity, pan_velocity, duration=0):
        """
        Send the command to tilt the camera by the specified number of degrees per s
econd in pan/tilt.
        This function has two modes.  First, if duration is 0, the initial velocity
is sent and
        then the function returns (meaning the camera will keep moving).  If duratio
n is greater than 0,
        the command executes for that amount of time and then sends a stop command t
o the camera
        and then returns.

        :param tilt_degrees: tile change in degrees per second
        :param pan_degrees: pan change in degrees per second
        :param duration: seconds to run the command for
        :return:
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Camera",
"Velocity")

        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
tilt_velocity, pan_velocity],
                                                            param_type_tuple=['float', '
float'], ack=False)


        if (duration > 0):
            # wait for the specified duration
            start_time = time.time()
            while (time.time() - start_time < duration):
                self.drone_connection.smart_sleep(0.1)

            # send the stop command
            self.drone_connection.send_param_command_packet(command_tuple, param_tup
le=[0, 0],
                                                                param_type_tuple=['float
', 'float'], ack=False)


    def set_max_altitude(self, altitude):
        """
        Set max altitude in meters.

        :param altitude: altitude in meters
        :return:
        """
        if (altitude < 0.5 or altitude > 150):
            print("Error: %s is not valid altitude. The altitude must be between 0.5
 and 150 meters" % altitude)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PilotingS
```

```
ettings", "MaxAltitude")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
altitude], param_type_tuple=['float'])

        while (not self.sensors.max_altitude_changed):
            self.smart_sleep(0.1)


    def set_max_distance(self, distance):
        """
        Set max distance between the takeoff and the drone in meters.

        :param distance: distance in meters
        :return:
        """
        if (distance < 10 or distance > 2000):
            print("Error: %s is not valid altitude. The distance must be between 10
and 2000 meters" % distance)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PilotingS
ettings", "MaxDistance")

        self.sensors.max_distance_changed = False

        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
distance], param_type_tuple=['float'])

        while (not self.sensors.max_distance_changed):
            self.smart_sleep(0.1)


    def enable_geofence(self, value):
        """
             If geofence is enabled, the drone won't fly over the given max distance
.
         1 if the drone can't fly further than max distance, 0 if no limitation on t
he drone should be done.

        :param value:
        :return:
        """
        if (value not in (0, 1)):
            print("Error: %s is not valid value. Valid value: 1 to enable geofence/
0 to disable geofence" % value)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PilotingS
ettings", "NoFlyOverMaxDistance")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
value], param_type_tuple=['u8'])

        while (not self.sensors.no_fly_over_max_distance_changed):
            self.smart_sleep(0.1)


    def set_max_tilt(self, tilt):
        """
        Set max pitch/roll in degrees

        :param tilt: max tilt for both pitch and roll in degrees
        :return:
        """
        if (tilt < 5 or tilt > 30):
            print("Error: %s is not valid tilt. The tilt must be between 5 and 30 de
grees" % tilt)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PilotingS
ettings", "MaxTilt")
```

```
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
tilt], param_type_tuple=['float'])

        while (not self.sensors.max_tilt_changed):
            self.smart_sleep(0.1)


    def set_max_tilt_rotation_speed(self, speed):
        """
        Set max pitch/roll rotation speed in degree/s

        :param speed: max rotation speed for both pitch and roll in degree/s
        :return:
        """
        if (speed < 80 or speed > 300):
            print("Error: %s is not valid speed. The speed must be between 80 and 30
0 degree/s" % speed)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "SpeedSett
ings", "MaxPitchRollRotationSpeed")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
speed], param_type_tuple=['float'])

        while (not self.sensors.max_pitch_roll_rotation_speed_changed):
            self.smart_sleep(0.1)


    def set_max_vertical_speed(self, speed):
        """
        Set max vertical speed in m/s

        :param speed: max vertical speed in m/s
        :return:
        """
        if (speed < 0.5 or speed > 2.5):
            print("Error: %s is not valid speed. The speed must be between 0.5 and 2
.5 m/s" % speed)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "SpeedSett
ings", "MaxVerticalSpeed")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
speed], param_type_tuple=['float'])

        while (not self.sensors.max_vertical_speed_changed):
            self.smart_sleep(0.1)


    def set_max_rotation_speed(self, speed):
        """
        Set max yaw rotation speed in degree/s

        :param speed: max rotation speed for yaw in degree/s
        :return:
        """
        if (speed < 10 or speed > 200):
            print("Error: %s is not valid speed. The speed must be between 10 and 20
0 degree/s" % speed)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "SpeedSett
ings", "MaxRotationSpeed")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
speed], param_type_tuple=['float'])

        while (not self.sensors.max_rotation_speed_changed):
            self.smart_sleep(0.1)
```

```
    def set_hull_protection(self, present):
        """
        Set the presence of hull protection - this is only needed for bebop 1
        1 if present, 0 if not present

        :param present:
        :return:
        """
        if (present not in (0, 1)):
            print("Error: %s is not valid value. The value must be 0 or 1" % present
)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "SpeedSett
ings", "HullProtection")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
present], param_type_tuple=['u8'])

        while (not self.sensors.hull_protection_changed):
            self.smart_sleep(0.1)


    def set_indoor(self, is_outdoor):
        """
        Set bebop 1 to indoor mode (not used in bebop 2!!)
        1 if outdoor, 0 if indoor

        :param present:
        :return:
        """
        if (is_outdoor not in (0, 1)):
            print("Error: %s is not valid value. The value must be 0 or 1" % is_outd
oor)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "SpeedSett
ings", "Outdoor")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
is_outdoor], param_type_tuple=['u8'])

        #while (not self.sensors.outdoor_mode_changed):
        #    self.smart_sleep(0.1)


    def set_picture_format(self, format):
        """
        Set picture format

        :param format:
        :return:
        """
        if (format not in ('raw', 'jpeg', 'snapshot', 'jpeg_fisheye')):
            print("Error: %s is not valid value. The value must be : raw, jpeg, snap
shot, jpeg_fisheye" % format)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "PictureFormatSelection", format)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.picture_format_changed):
            self.smart_sleep(0.1)


    def set_white_balance(self, type):
        """
        Set white balance

        :param type:
```

```
        :return:
        """
        if (type not in ('auto', 'tungsten', 'daylight', 'cloudy', 'cool_white')):
            print("Error: %s is not valid value. The value must be : auto, tungsten,
 daylight, cloudy, cool_white" % type)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "AutoWhiteBalanceSelection", type)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.auto_white_balance_changed):
            self.smart_sleep(0.1)


    def set_exposition(self, value):
        """
        Set image exposure

        :param value:
        :return:
        """
        if (value < -1.5 or value > 1.5):
            print("Error: %s is not valid image exposure. The value must be between
-1.5 and 1.5." % value)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PictureSe
ttings", "ExpositionSelection")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
value], param_type_tuple=['float'])

        while (not self.sensors.exposition_changed):
            self.smart_sleep(0.1)


    def set_saturation(self, value):
        """
        Set image saturation

        :param value:
        :return:
        """
        if (value < -100 or value > 100):
            print("Error: %s is not valid image saturation. The value must be betwee
n -100 and 100." % value)
            print("Ignoring command and returning")
            return

        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PictureSe
ttings", "SaturationSelection")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
value], param_type_tuple=['float'])

        while (not self.sensors.saturation_changed):
            self.smart_sleep(0.1)


    def set_timelapse(self, enable, interval=8):
        """
        Set timelapse mode

        :param enable:
        :param interval:
        :return:
        """
        if (enable not in (0, 1) or interval < 8 or interval > 300):
            print("Error: %s or %s is not valid value." % (enable, interval))
            print("Ignoring command and returning")
            return
```

```
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "PictureSe
ttings", "TimelapseSelection")
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple=[
enable, interval], param_type_tuple=['u8', 'float'])

        while (not self.sensors.timelapse_changed):
            self.smart_sleep(0.1)


    def set_video_stabilization(self, mode):
        """
        Set video stabilization mode

        :param mode:
        :return:
        """
        if (mode not in ('roll_pitch', 'pitch', 'roll', 'none')):
            print("Error: %s is not valid value. The value must be : roll_pitch, pit
ch, roll, none" % mode)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "VideoStabilizationMode", mode)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.video_stabilization_changed):
            self.smart_sleep(0.1)


    def set_video_recording(self, mode):
        """
        Set video recording mode

        :param mode:
        :return:
        """
        if (mode not in ('quality', 'time')):
            print("Error: %s is not valid value. The value must be : quality, time"
% mode)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "VideoRecordingMode", mode)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.video_recording_changed):
            self.smart_sleep(0.1)


    def set_video_framerate(self, framerate):
        """
        Set video framerate

        :param framerate:
        :return:
        """
        if (framerate not in ('24_FPS', '25_FPS', '30_FPS')):
            print("Error: %s is not valid value. The value must be : 24_FPS, 25_FPS,
 30_FPS" % framerate)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "VideoFramerate", framerate)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.video_framerate_changed):
```

```
            self.smart_sleep(0.1)


    def set_video_resolutions(self, type):
        """
        Set video resolutions

        :param type:
        :return:
        """
        if (type not in ('rec1080_stream480', 'rec720_stream720')):
            print("Error: %s is not valid value. The value must be : rec1080_stream4
80, rec720_stream720" % type)
            print("Ignoring command and returning")
            return

        (command_tuple, enum_tuple) = self.command_parser.get_command_tuple_with_enu
m("ardrone3", "PictureSettings", "VideoResolutions", type)
        self.drone_connection.send_enum_command_packet_ack(command_tuple, enum_tuple
)

        while (not self.sensors.video_resolutions_changed):
            self.smart_sleep(0.1)


############################################################

# Author: Ronan Donohue
# Number: c00208501

    def move_to(self, lat, lng, altitude, orientation, heading):
        """
        This function will move the drone to a specific location.
        If a new command move_to is sent, the drone will immediatly run it (no cance
l will be issued).
        If a cancel_move_to is sent, the move_to is stopped.

        :param lat: latitude of the location (in degrees) to reach
        :param long: longitude of the location (in degrees) to reach
        :param altitude: altitude above sea level (in m) to reach
        :param orientation: Orientation mode of the move to, string, either
            'NONE', the drone won't change its orientation
            'TO_TARGET'; the drone will make a rotation to look in direction of the
given location
            'HEADING_START';
                The drone will orientate itself to the given heading before moving t
o the location
            'HEADING_DURING';
                The drone will orientate itself to the given heading while moving to
 the location
        :param heading: Heading (relative to the North in degrees).
            This value is only used if the orientation mode is HEADING_START or HEAD
ING_DURING
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "moveTo")
        if orientation > 3 or orientation < 0:
            orientation = 0
        param_tuple = [lat, lng, altitude, orientation, heading]
        param_type_tuple = ['float', 'float', 'float', 'str', 'float']
        self.drone_connection.send_param_command_packet(command_tuple, param_tuple,
param_type_tuple)


    def cancel_move_to(self):
        """
        Cancel the current move_to
        If there is no current move_to, this command has no effect.
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "CancelMoveTo")
        self.drone_connection.send_noparam_command_packet_ack(command_tuple) # High
Priority sending an option
```

```
    def return_home(self):
        """
        Call the drone back to its starting location
        """
        command_tuple = self.command_parser.get_command_tuple("ardrone3", "Piloting"
, "NavigateHome")
        self.drone_connection.send_noparam_command_packet_ack(command_tuple)


    def get_current_location(self):
        """
        Returns a tuple containing the current lat, lng, and alt
        of the drone
        """
        lat = self.sensors.sensors_dict['GpsLocationChanged_latitude']
        lng = self.sensors.sensors_dict['GpsLocationChanged_longitude']
        alt = self.sensors.sensors_dict['GpsLocationChanged_altitude']
        return (lat, lng, alt)


    def is_hovering(self):
        """
        Returns true if it is landed or emergency and False otherwise
        :return: bool
        """
        if (self.sensors.flying_state in ("hovering")):
            return True
        else:
            return False


    def reached_location(self):
        """
        Returns True if the drone has reached its location,
        False otherwise
        """
        if (self.sensors.move_status in ("DONE")):
            return True
        else:
            return False
```

```
//Author: Ronan Donohue
//Student Number: c00208501

import QtQuick 2.7
import QtQml 2.5
import QtQuick.Controls 1.3
import QtQuick.Controls.Styles 1.3
import QtQuick.Window 2.2
import QtQuick.Layouts 1.2
import QtPositioning 5.9
import QtLocation 5.6
import QtQuick.Dialogs 1.1

////////////////////////////////////////////////////////////////////////////////
/*
* This first section contains all the code that governs the layout of the
* main application window
*/

ApplicationWindow {
    /*
     * Main Application window for DATCS
     */

    id: root
    width: 1200
    height: 700
    visible: true
    property var countOfClicks: 0
    property int dronesAdded: 0
    property bool simDrone: false
    property bool realDrone: false

    ListModel {
        id: droneMarkers
    }

    ListModel {
        id: geoPoints
    }

    ListModel {
        id: planNames
    }

    ListModel {
        id: droneNames
    }

////////////////////////////////////////////////////////////////////////////////
/*
* This section contains all the code neccessary for the menu
*/
    menuBar: MenuBar {
        /*
         * The Menu displays options to the user that, when clicked,
         * will invoke QML functions to swap in/out various
         * elements to the main window
         */

        id: menuBar
        Menu {
            id: droneMenu
            title: qsTr("&Drone")
            MenuItem {
                text: qsTr("&New Sim Drone")
                onTriggered: {
                    root.addNewSimDrone()
                }
            }
            MenuItem {
                text: qsTr("&New Bebop Drone")
                onTriggered: {
```

```
                    root.addNewRealDrone()
                }
            }
            MenuItem {
                text: qsTr("&Quit")
                onTriggered: {
                    if (root.realDrone = true) {
                        atc.shutdown_feed_and_drone()
                    }
                    Qt.quit();
                }
            }
        }
        Menu {
            id: flightPlanMenu
            title: qsTr("&Flight Plan")
            MenuItem {
                text: qsTr("&New Flight Plan")
                onTriggered: {
                    root.createNewFlightPlan()
                }
            }
        }
        Menu {
            title: qsTr("&Help")
            MenuItem {
                text: qsTr("&About")
                onTriggered: {
                    root.displayInformation()
                }
            }
        }
    }

//////////////////////////////////////////////////////////////////////////////////
/*
* All things Map related go here
*/

    Plugin {
        /*
         * Map Plugin
         */
        id: mapPlugin
        name: "osm" //"mapboxgl" "osm" "esri"
    }

    Map {
        /*
         * Map Item, displays a map using mapPlugin
         */
        id: map
        anchors.fill: parent
        plugin: mapPlugin
        center: atc.location
        copyrightsVisible: false
        zoomLevel: 14


        MapPolyline {
            /*
             * A Drone flight path visualised
             */
            id: fp_line
            line.width: 2
            line.color: 'blue'
            path: []
        }

        Rectangle {
            id: infoBox
            anchors.centerIn: parent
            color: "white"
```

```
            border.width: 1
            width: text.width * 1.3
            height: text.height * 1.3
            radius: 5
            Text {
                id: text
                anchors.centerIn: parent
                text: qsTr("Welcome to DATCS!")
            }

            Timer {
                interval: 5000; running: true; repeat: false;
                onTriggered: fadeOut.start()
            }

            NumberAnimation {
                id: fadeOut; target: infoBox;
                property: "opacity";
                to: 0.0;
                duration: 200
                easing.type: Easing.InOutQuad
            }
        }

////////////////////////////////////////////////////////////////////////////////
/*
* This section contains everything about the changing MouseArea functionality
* for the Map between various menu options. It is a child of the Map item
* because I didn't want to override basic Mouse functionality in any of the
* parent Items.
*/
        MouseArea {
            /*
            * Change mouse functionality for Map area
            */

            id: mousearea
            anchors.fill: map
            acceptedButtons: Qt.LeftButton | Qt.RightButton
            hoverEnabled: true
            property var coord: map.toCoordinate(Qt.point(mouseX, mouseY))

            Drone {   }

            Label {
                id: placeDroneHere
                visible: false
                x: parent.mouseX - width
                y: parent.mouseY - height - 5
                text: "Double Click to place Drone: " + droneName.text
            }

            Label {
                id: latLngInfo
                visible: false
                x: parent.mouseX - width
                y: parent.mouseY - height - 5
                text: "lat: %1; lon: %2".arg(parent.coord.latitude).arg(parent.coord
.longitude)
            }

            onDoubleClicked: {
                if (mouse.button === Qt.LeftButton && placeDroneHere.visible === tru
e)
                {
                    var newDrone = mapDroneComponent.createObject(map)
                    atc.add_drone(droneName.text, '' + mousearea.coord.latitude, ''
+ mousearea.coord.longitude, "Sim")
                    newDrone.setName(droneName.text)
                    newDrone.updateDrone(atc.get_drone_location(newDrone.getName()))
                    map.addMapItem(newDrone)
                    root.dronesAdded += 1
                    placeDroneHere.visible = false
```

```
                    droneName.text = ""
                }
                else if (mouse.button === Qt.LeftButton && latLngInfo.visible == tru
e)
                {
                    undoButton.enabled = true
                    //Add coordinate to MapPolyline
                    fp_line.insertCoordinate(countOfClicks, coord)
                    map.addMapItem(fp_line) //this will only run once if the item is
 already present on the map
                    //Increment number of double clicks
                    countOfClicks = countOfClicks + 1
                    //Add to ListModel
                    geoPoints.append({
                        "point": countOfClicks,
                        "latitude": coord.latitude,
                        "longitude": coord.longitude,
                        "altitude": 0.0})
                        //Display to user
                    //Display to user
                    flightPlanInfo.append("Point: " + countOfClicks + ", Lat: " + co
ord.latitude + ", Longitude: " + coord.longitude)
                }
            }
        }
    }
////////////////////////////////////////////////////////////////////////////////
/*
* This section contains code responsible for the various MeesageDialogs that
* are displayed to the user
*/

    MessageDialog {
        /*
         * Ask user for confirmation regarding proposed flight plan
         */

        id: confirmDialog
        icon: StandardIcon.Question
        title: "Complete Proposed Flight Plan?"
        text: "Are you sure that you want to save the proposed flight plan?"
        standardButtons: StandardButton.Ok | StandardButton.Cancel

        onAccepted: {
            atc.create_flight_plan(fpName.text)
            for (var i = 0; i < geoPoints.count; i++)
            {
                var lat = '' + geoPoints.get(i).latitude
                var lng = '' + geoPoints.get(i).longitude
                var alt = '' + geoPoints.get(i).altitude
                atc.append_flight_plan_coordinates(fpName.text, lat, lng, alt)
            }
            flightPlanInfo.text = qsTr("--Double-click the map to enter a coordinate
--")
            countOfClicks = 0
            root.removeMapPolylinePoints()
            map.removeMapItem(fp_line)
            geoPoints.clear()
        }

        onRejected: {
            flightPlanInfo.text = qsTr("--Double-click the map to enter a coordinate
--")
            countOfClicks = 0
            fp_line.path = []
            root.removeMapPolylinePoints()
            map.removeMapItem(fp_line)
            geoPoints.clear()
        }
    }

    MessageDialog {
        id: pleaseEnterFPName
```

```
        icon: StandardIcon.Warning
        title: "Flight Plan Needs A Name!"
        text: "You must enter a Flight Plan name to proceed"
        standardButtons: StandardButton.Ok
        onAccepted: {
            pleaseEnterFPName.close()
            addName.requestActivate()
        }
    }

    MessageDialog {
        id: flightPlanAssigned
        icon: StandardIcon.Warning
        title: "Flight plan already assigned"
        text: ""
        standardButtons: StandardButton.Ok
        onAccepted: {
            flightPlanAssigned.close()
        }
    }

    MessageDialog {
        id: noPointsDialog
        icon: StandardIcon.Warning
        title: "No Points Added"
        text: "You need to set some points first!"
        standardButtons: StandardButton.Ok
        onAccepted: {
            noPointsDialog.close()
        }
    }

    MessageDialog {
        id: noAltSetDialog
        icon: StandardIcon.Warning
        title: "Altitudes Cannot be 0"
        text: "Set your altitudes to values other than 0"
        standardButtons: StandardButton.Ok
        onAccepted: {
            noAltSetDialog.close()
            altWindow.requestActivate()
        }
    }

    MessageDialog {
        id: droneAdded
        width: root.width
        icon: StandardIcon.Information
        title: "Drone Air Traffic Control System"
        text: "Drone " + droneName.text + " added."
        informativeText: ""
        standardButtons: StandardButton.Ok
        onAccepted: {
            placeDroneHere.visible = false
            droneName.text = ""
        }
    }

    MessageDialog {
        id: nameTaken
        icon: StandardIcon.Warning
        title: "Drone Air Traffic Control System"
        text: "Drone name " + droneName.text + " is already in use!"
        standardButtons: StandardButton.Ok
        onAccepted: {
            nameTaken.close()
        }
    }

////////////////////////////////////////////////////////////////////////////////
/*
* This section contains various ColumnLayouts, Windows and RowLayouts that
* change as the program executes
```

```
*/

    ColumnLayout {
        id: addCoordinateInfo
        anchors.right: parent.right
        anchors.top: parent.top
        visible: false
        RowLayout {
            TextArea {
                id: flightPlanInfo
                height: 100
                text: qsTr("--Double-click the map to enter a coordinate--")
                readOnly: true
            }
        }

        RowLayout {
            anchors.right: parent.right
            anchors.bottom: parent.bottom
            Button {
                text: "Cancel"
                onClicked: {
                    flightPlanInfo.text = qsTr("--Double-click the map to enter a co
ordinate--")
                    countOfClicks = 0
                    geoPoints.clear()
                    fp_line.path = []
                    root.removeMapPolylinePoints()
                    map.removeMapItem(fp_line)
                    root.closeAll()
                }
            }
            Button {
                id: undoButton
                text: "Undo"
                onClicked: {
                    var inx = countOfClicks - 1
                    if (inx < 0) {
                        undoButton.enabled = false
                    }
                    else {
                        fp_line.removeCoordinate(inx)
                        geoPoints.remove(inx, 1)
                        flightPlanInfo.undo()
                        countOfClicks = countOfClicks - 1
                    }
                }
            }
            Button {
                id: nextButton
                text: "Next"
                onClicked: {
                    if (geoPoints.count == 0)
                    {
                        noPointsDialog.open()
                    }
                    else
                    {
                        altWindow.show()
                    }
                }
            }
        }
    }

    Window {
        id: addName
        title: "Add Altitudes"
        width: 200
        height: 200
        color: "gainsboro"

        ColumnLayout {
```

```
            id: addNameColumn

            ColumnLayout {
                id: addNameRow

                Label {
                    id: promptFPName
                        text: "Enter Flight Plan name:"
                    }

                TextField {
                    id: fpName
                    focus: true
                    text: ""
                }
            }

            RowLayout {
                Button {
                    id: notSoSure
                    text: "Back"
                    onClicked: {
                        fpName.text = ""
                        addName.hide()
                        altWindow.show()
                    }
                }

                Button {
                    id: areWeSure
                    text: "Done"
                    onClicked: {
                        if (root.isFlightPlanNameBlank())
                        {
                            pleaseEnterFPName.open()
                        }
                        else
                        {
                            confirmDialog.open()
                            addName.hide()
                        }
                    }
                }
            }
        }
    }

    Window {
        id: altWindow
        title: "Add Altitudes"
        width: root.getWidth()
        height: 200
        modality: Qt.WindowModal
        color: "gainsboro"

        ColumnLayout {
            width: parent.width
            spacing: 2
            RowLayout {
                spacing: 2

                Label {
                    width: parent.width
                    height: 10
                    text: qsTr("Please set the altitude in meters \nfor each coordin
ate:")
                }
            }

            RowLayout {
                width: parent.width
                height: 100
```

```
            Repeater {
                model: geoPoints.count
                delegate: Column {
                    id: cid
                    width: 40
                    height: parent.height
                    //Bebop 2's wifi can cut out over
                    //300 meters, playing it safe with 280
                    property real maxHeight: 280.0

                    //cValue set to max
                    //sets the slider to 0 oddly enough
                    property real cValue: cid.maxHeight

                    Text {
                        id: textID
                        text: cid.cValue
                        height: 25
                    }

                    Item {
                        //spacer item
                        width: cid.width
                        height: 5
                    }

                    Slider {
                        id: sliderID
                        width: cid.width
                        height: 100
                        orientation: Qt.Vertical
                        maximumValue: cid.maxHeight
                        value: cid.cValue
                        stepSize: 0.25
                        onValueChanged: {
                            cid.cValue = sliderID.value
                            geoPoints.setProperty(index, "altitude", cid.cValue)
                        }
                    }
                }
            }
        }

        RowLayout {
            spacing: 2

            Button {
                text: "Back"
                onClicked: {
                    altWindow.hide()
                }
            }

            Button {
                text: "Add Name"
                onClicked: {
                    if (root.areAltitudesSet() == false)
                    {
                        noAltSetDialog.open()
                    }
                    else
                    {
                        altWindow.hide()
                        addName.show()
                    }
                }
            }
        }
    }
}

Window {
    id: getDroneName
```

```
        width: 200
        height: 90
        visible: false
        modality: Qt.WindowModal
        color: "gainsboro"

        ColumnLayout {
            spacing: 4
            Label {
                text: "Enter the new drones name:"
            }

            TextField {
                id: droneName
                focus: true
                text: ""
            }

            Button {
                text: "Done"
                onClicked: {
                    if (root.checkDroneName(droneName.text) == false) {
                        getDroneName.close()
                        if (root.simDrone == true) {
                            placeDroneHere.visible = true
                        }
                        else {
                            atc.add_drone(droneName.text, '' + 0.0, '' + 0.0, "real"
)

                            var droneLocation = atc.get_real_drone_location()
                            var newDrone = mapDroneComponent.createObject(map)
                            newDrone.setName(droneName.text)
                            newDrone.setColor('green')
                            newDrone.updateDrone(atc.get_drone_location(newDrone.get
Name()))

                            newDrone.enableFeedView()
                            newDrone.enableRunTest()
                            newDrone.enableLandDrone()
                            map.addMapItem(newDrone)
                            root.realDrone = true
                            root.dronesAdded += 1
                            droneName.text = ""
                        }
                    }
                    else {
                        nameTaken.open()
                        droneName.text = ""
                    }
                }
            }
        }
    }

    Window {
        id: showFlightPlans
        title: "Select a Flight Plan"
        width: 250
        height: 300

        TableView {
            id: nameView
            width: parent.width
            height: parent.height
            model: planNames

            TableViewColumn {
                role: "Name"
                title: "Flight Plan Name"
            }
        }
    }

    Window {
```

```
        id: showHelp
        width: 640
        title: "Help"
        height: 480
        color: 'gainsboro'
        ColumnLayout {
            spacing: 2
            Text {
                id: helpText
                width: showHelp.width
                height: showHelp.height
                textFormat: Text.StyledText
                horizontalAlignment: Text.AlignHCenter
                text: "<h1>Welcome to Drone Air Traffic Control System</h1>
                <br>
                <h3>Creating a Flight Plan</h3><br>
                <p>To create a flight plan, select <i>Flight Plan</i> ->
                <i>New Flight Plan</i><br>
                To place a coordinate for your flight plan, <br>
                double-click the screen.
                If you make a mistake, you can click <i>Undo</i> or if you
                wish to return, click <i>Cancel</i></p><br>
                <h3>Creating a Simulated Drone</h3><br>
                <p>To create a simulated drone, select <i>Drone</i> ->
                <i>New Sim Drone</i>.<br>
                To place your simulated drone, select a point on the map and
                double click.<br>
                To assign a flight plan to your drone, right-click
                the drone icon<br>
                and select <i>'Assign Flight Plan'</i>. Once the
                flight plan has been assigned, <br>
                to have the drone run the flight
                plan you must click on the drone.</p>
                <h3>Creating a Real Drone</h3>
                <p>To create a real drone, first, make sure you are connected
                <br>
                to the drone's wifi and click <i>Drone</i> -> <i>New Real Drone </i>
                <br>
                After a short while, the drone will display on the map as a Green ci
rcle. <br>
                You can right-click the drone to perform similar actions to the simu
lated drone.<p>"
            }
        }
    }

    ScrollView {
        id: atcScroll
        anchors.left: parent.left
        anchors.bottom: parent.bottom
        width: root.width / 4
        height: root.height / 3
        TextArea {
            id: atcText
            readOnly: true
            wrapMode: TextEdit.Wrap
            width: root.width / 4
            height: root.height / 3
            style: TextAreaStyle {
                textColor: 'white'
                backgroundColor: 'black'

            }
            text: "Welcome to Drone Air Traffic Control System"
        }
    }

////////////////////////////////////////////////////////////////////////////////
/*
* QML functions
*/
    function createNewFlightPlan()
    {
```

```
        root.closeAll()
        addCoordinateInfo.visible = true
        latLngInfo.visible = true
    }

    function establishDroneCorridor()
    {
        root.closeAll()
    }

    function addNewSimDrone()
    {
        root.closeAll()
        root.simDrone = true
        getDroneName.show()
    }

    function addNewRealDrone()
    {
        root.closeAll()
        getDroneName.show()
    }

    function setCourse()
    {
        root.closeAll()
    }

    function displayInformation()
    {
        root.closeAll()
        showHelp.show()
    }

    function closeAll()
    {
        hiddenRect.droneFPAssignmentCheck = false
        placeDroneHere.visible = false
        addCoordinateInfo.visible = false
        latLngInfo.visible = false
        getDroneName.close()
        planNames.clear()
        showFlightPlans.close()
        showHelp.close()
        //getDroneCorridor.close()
    }

    function checkDroneName(name)
    {
        return atc.is_name_taken(name)
    }

    function removeMapPolylinePoints()
    {
        /*
         * Removes drone flight plan points from user display
         */
        for (var i = 0; i <= fp_line.pathLength(); ++i)
        {
            fp_line.removeCoordinate(i)
        }
        //loop needs to run again as MapPolyline will retain one point in
        //path regardless of the number of removeCoordinate() calls made
        for (var i = 0; i <= fp_line.pathLength(); i++)
        {
            fp_line.removeCoordinate(i)
        }
    }

    function getWidth()
    {
        var countOfPoints = geoPoints.count
        var colWidth = 50
```

```
        var retVal = 250
        if (countOfPoints > 5)
        {
            retVal = countOfPoints * colWidth
        }
        return retVal
    }

    function areAltitudesSet()
    {
        for (var i = 0; i < geoPoints.count; ++i)
        {
            if (geoPoints.get(i).altitude == 0.0)
            {
                return false
            }
        }
        return true
    }

    function isFlightPlanNameBlank()
    {
        if (fpName.text == "")
        {
            return true
        }
        return false
    }

    function displayNames(names)
    {
        for (var i = 0; i < names.length; ++i)
        {
            planNames.append({"name": names[i]['flight_plan_name']})
        }
    }

////////////////////////////////////////////////////////////////////////////////
/*
* This section contains all Connections needed to
* send/receive Signals
*/

    Connections {
        target: atc
        onListFPNames: {
            root.displayNames(flightPlanNames)
        }

        /*onSendDroneInfo: {
            //atcText.append("Drone info requested for: " + requestingDrone)
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].droneName == requestingDrone) {
                    map.mapItems[i].sendDroneInfo()
                }
            }
        }*/

        onMoveDrone: {
            //atcText.append("moveDrone signal received")
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].droneName == thisDroneName) {
                    //atcText.append(map.mapItems[i].getName())
                    map.mapItems[i].flyMyPretties(currentLocation)
                }
            }
        }

        onStopDrone: {
            atcText.append("stopDrone signal received")
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].droneName == stopDroneName) {
                    //map.mapItems[i].pause()
```

```
                    atcText.append(stopDroneName, " needs to stop")
                }
            }
        }

        onPlotDroneRouteViz: {
            var droneRoute = droneRouteViz.createObject(map)

            droneRoute.line.width = 2
            droneRoute.line.color = 'red'
            droneRoute.objectName = flight_plan_name
            for (var i = 0; i < flight_plan_coordinates.length; ++i)
            {
                droneRoute.addCoordinate(flight_plan_coordinates[i])
            }
            map.addMapItem(droneRoute)
        }

        onRemoveDronePath: {
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].objectName == path_name) {
                    map.removeMapItem(map.mapItems[i])
                }
            }
        }

        onUpdateATCConsole: {
            atcText.append(update)
        }

        onMovingHome: {
            atcText.append("Drone" + offhome + " is heading home")
        }

        onArrivedHome: {
            atcText.append(home)
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].droneName == homeDrone) {
                    map.mapItems[i].arrivedHome()
                }
            }
        }

        onFlightPlanAlreadyAssigned: {
            flightPlanAssigned.text = "Flight plan " + assignedFPName + " has been a
ssigned already."
            flightPlanAssigned.open()
        }

        onFlightPlanFinished: {
            for (var i = 0; i < map.mapItems.length; ++i) {
                if (map.mapItems[i].droneName == droneFinishedFP) {
                    map.mapItems[i].notRunning()
                }
            }
        }
    }

    Connections {
        /*
         * This connection is for the TableView
         * containing the Flight Plan names.
         */

        target: nameView
        onClicked: {
            var fpname = planNames.get(row).name
            for (var i = 0; i < map.mapItems.length; ++i)
            {
                if (map.mapItems[i].droneName == hiddenRect.currentDroneSelectedName
)
                {
                    atcText.append(map.mapItems[i].getName())
```

```
                    map.mapItems[i].setFlightPlan(fpname)
                }
            }
        }
    }


//////////////////////////////////////////////////////////////////////////////
/*
 * This section contains miscellaneous components used to create
 * custom Python/QML objects in the main view
 */
    Component {
        /*
         *
         */
        id: mapDroneComponent
        Drone {}
    }

    Component {
        id: droneRouteViz
        MapPolyline {}
    }

    Rectangle {
        id: hiddenRect
        property bool droneFPAssignmentCheck: false
        property string currentDroneSelectedName: ""
        visible: false
    }
}
```

```
import QtQuick 2.7
import QtQml 2.5
import QtQuick.Controls 1.3
import QtQuick.Controls.Styles 1.3
import QtQuick.Window 2.2
import QtQuick.Layouts 1.2
import QtPositioning 5.9
import QtLocation 5.6
import QtQuick.Dialogs 1.1

// Drone.qml
MapQuickItem {
    id: drone
    // Must be instantiated for MapQuickItem to work
    coordinate: QtPositioning.coordinate(0.0, 0.0, 0.0)
    property string droneName
    property string fpName
    property bool fpSet: false
    property bool running: false
    property bool returningHome: false

    anchorPoint.x: blip.width/2
    anchorPoint.y: blip.height/2

    sourceItem: Grid {
        columns: 1
        Grid {
            horizontalItemAlignment: Grid.AlignHCenter

            Rectangle {
                id: blip
                width: 25
                height: 25
                color: "red"
                border.color: "black"
                border.width: 1
                radius: 50
            }

            Rectangle {
                id: bubble
                color: "lightblue"
                border.width: 1
                width: text.width * 1.3
                height: text.height * 1.3
                radius: 5
                Text {
                    id: text
                    anchors.centerIn: parent
                    text: droneName
                }
            }
        }

        Rectangle {
            id: message
            color: "lightblue"
            border.width: 1
            width: banner.width * 1.3
            height: banner.height * 1.3
            radius: 5
            opacity: 0
            Text {
                id: banner
                anchors.centerIn: parent
            }
            SequentialAnimation {
                id: playMessage
                running: false
                NumberAnimation { target: message;
                    property: "opacity";
                    to: 1.0;
                    duration: 2000
```

```
                easing.type: Easing.Linear
            }
            PauseAnimation  { duration: 1000 }
            NumberAnimation { target: message;
                property: "opacity";
                to: 0.0;
                duration: 2000}
        }
    }
}

MouseArea {
    anchors.fill: parent
    acceptedButtons: Qt.LeftButton | Qt.RightButton

    onClicked: {
        hiddenRect.currentDroneSelectedName = drone.droneName

        if (mouse.button === Qt.LeftButton) {
            if (fpSet == false){
                noFPSet.open()
            }
        }
        if (mouse.button === Qt.RightButton) {
            if (drone.returningHome == true) {
                retHomeMenuOption.enabled = false //No returning home
            }
            else {
                retHomeMenuOption.enabled = true
            }
            if (drone.running == true) {
                assignPlanMenuOption.enabled = false //No assigning flight plans

            }
            else {
                assignPlanMenuOption.enabled = true
            }
            atc.list_flight_plans()
            contextMenu.popup()
        }
    }

    onDoubleClicked: {
        if (mouse.button === Qt.LeftButton)
        {
            atc.begin_flight_plan(drone.droneName)
            drone.running = true
        }
    }
}

MessageDialog {
    id: noFPSet
    icon: StandardIcon.Warning
    title: "Drone Air Traffic Control System"
    text: "You need to assign a flight plan to " + qsTr(droneName) + " before it
can begin a flight"
    standardButtons: StandardButton.Ok
    onAccepted: {
        noFPSet.close()
    }
}

MessageDialog {
    id: droneGoingHome
    icon: StandardIcon.Information
    text: "Drone Air Traffic Control System"
    title: "Drone " + drone.droneName + " is already going home!"
    standardButtons: StandardButton.Ok
    onAccepted: {
        droneGoingHome.close()
    }
}
```

```
    MessageDialog {
        id: droneDelete
        icon: StandardIcon.Question
        title: "Confirm?"
        text: "Are you sure you want to delete " + drone.droneName + "?"
        standardButtons: StandardButton.Ok | StandardButton.Cancel
        onAccepted: {
            atc.delete_drone(drone.droneName)
            map.removeMapItem(drone)
        }
        onRejected: {
            droneDelete.close()
        }
    }

    MessageDialog {
        id: assignFPtoDrone
        icon: StandardIcon.Question
        title: "Confirm?"
        text: "Assign flight plan " + drone.fpName + " to drone " + drone.droneName
+ "?"
        standardButtons: StandardButton.Ok | StandardButton.Cancel
        onAccepted: {
            fpSet = true
            atc.assign_flight_plan_to_drone(drone.droneName, drone.fpName)
            hiddenRect.droneFPAssignmentCheck = false
            showFlightPlans.hide()
        }
        onRejected: {
            drone.fpName = ""
            hiddenRect.droneFPAssignmentCheck = false
            assignFPtoDrone.close()
        }

    }

    Menu {
        id: contextMenu
        visible: false

        MenuItem {
            id: assignPlanMenuOption
            text: "Assign a Flight Plan"
            enabled: true
            onTriggered: {
                showFlightPlans.show()
            }
        }

        MenuItem {
            id: speedUpMenuOption
            text: "Speed Up"
            enabled: false
            onTriggered: {
                atc.speedup(drone.droneName)
            }
        }

        MenuItem {
            id: slowDownMenuOption
            text: "Slow Down"
            enabled: false
            onTriggered: {
                atc.slowdown(drone.droneName)
            }
        }

        MenuItem {
            id: retHomeMenuOption
            text: "Return Home"
            onTriggered: {
                if (drone.returningHome == false) {
```

```
                    atc.call_drone_home(drone.droneName)
                    drone.returningHome = true
                }
                else {
                    droneGoingHome.open()
                }
            }
        }

        MenuItem {
            id: delDroneMenuOption
            text: "Delete Drone"
            onTriggered: {
                droneDelete.open()
            }
        }

        MenuItem {
            id: viewDroneCameraFeedMenuOption
            text: "View Drone Camera Feed"
            enabled: false
            onTriggered: {
                atc.fork_vlc_and_view_drone_feed(drone.droneName)
            }
        }

        MenuItem {
            id: runTestMenuOption
            text: "Run Test"
            enabled: false
            onTriggered: {
                atc.run_test(drone.droneName)
            }
        }

        MenuItem {
            id: landDroneMenuOption
            text: "Land Drone"
            enabled: false
            onTriggered: {
                atc.land_drone(drone_name)
            }
        }
    }

    function updateDrone(newCoord) {
        drone.coordinate = newCoord
    }

    /*function sendDroneInfo() {
        atc.process_messages(drone.droneName ,drone.coordinate)
    }*/

    function setName(name) {
        drone.droneName = name
    }

    function getName() {
        return drone.droneName
    }

    function setFlightPlan(name) {
        var fpAssigned = atc.is_flight_plan_already_assigned(name)
        if (fpAssigned == false) {
            drone.fpName = name
            assignFPtoDrone.open()
        }
        else {
            flightPlanAssigned.text = "Flight plan " + name + " is already assigned"
            flightPlanAssigned.open()
        }
    }
```

```
    function flyMyPretties(currentLocation) {
        drone.running = true
        speedUpMenuOption.enabled = true
        slowDownMenuOption.enabled = true
        drone.coordinate = currentLocation
    }

    function arrivedHome() {
        drone.returningHome = false
        speedUpMenuOption.enabled = false
        slowDownMenuOption.enabled = false
        assignPlanMenuOption.enabled = true
        drone.running = false
    }

    function notRunning() {
        drone.running = false
        speedUpMenuOption.enabled = false
        slowDownMenuOption.enabled = false
    }

    function setColor(newColor){
        blip.color = newColor
    }

    function enableFeedView() {
        viewDroneCameraFeedMenuOption.enabled = true
    }

    function enableRunTest() {
        runTestMenuOption.enabled = true
    }

    function enableLandDrone() {
        landDroneMenuOption.enabled = true
    }
}
```

```
# Author: Ronan Donohue
# Number: c00208501

__license__ = "MIT"
__revision__ = "DroneController.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'

import functools
import time

from PyQt5.QtPositioning import QGeoCoordinate
from PyQt5.QtCore import (QObject, QTimer, pyqtSignal,
        pyqtSlot, pyqtProperty, QTime)

from DATCS_Bebop import DATCS_Bebop

from SimulatedDrone import SimulatedDrone

class DroneController(QObject):
    """
    This class is responsible for 'piloting' the drones it connects to, simulated
    or real.

    :param name: str, the name of the DroneController
    :param home: a QGeoCoordinate location containing the home location
    for the DroneController object
    :param which_type: str, describing the type of drone the DroneController will pi
lot.
    """

    def __init__(self, name, home, which_type, parent=None):
        """
        Initializes the DroneController object
        """
        super().__init__(parent)
        self.name = name
        self.possible_states = tuple(["landed", "taking off",
                                      "flying", "emergency", "landing", "hovering"])
        self.current_state = self.possible_states[
            self.possible_states.index("landed")]
        self.bearing_reading = 0.0
        self.altitude_reading = 0.0
        self.latitude_reading = 0.0
        self.longitude_reading = 0.0
        self.battery_level = 100

        self.home = home
        self.current_location = home
        self.point_moving_towards = QGeoCoordinate()

        self.speed = 20.00 #Default for SimDrone
        self.time_spent_travelling = 0
        self.time_spent_travelling_towards_home = 0
        self.next_point = QGeoCoordinate()

        self.is_active = False
        self.is_moving = False
        self.is_waiting = False
        self.route_complete = False
        self.moving_home = False
        self.assigned_flight_plan = ""
        self.arrived_at_start_point = False
        self.drone = None
        self.which_type = which_type

        if self.which_type == "Sim":
            self.drone = SimulatedDrone(home)
        else:
            # You need to be connected to the drone wifi beforehand
            self.drone = DATCS_Bebop("Bebop2")
            print("connecting")
            success = self.drone.connect(10) # this can cause slow down in GUI
            print(success)
```

```
            #self.drone.smart_sleep(5)
            self.drone.ask_for_state_update()
            self.set_current_location()
            self.home = self.current_location

###############################################################################


    def update_readings(self):
        """
        Gets the current location from the drone and updates each lat, lng
        and alt reading for the DroneController
        """
        self.set_current_location()
        self.latitude_reading = self.current_location.latitude()
        self.longitude_reading = self.current_location.longitude()
        self.altitude_reading = self.current_location.altitude()


    def takeoff(self):
        """
        Sets the DroneController active status to true, so now the AirTrafficControl
ler can begin
        polling it. Depending on the type of drone the DroneController is piloting,
a takeoff command
        is sent.
        """
        self.is_active = True
        if self.which_type == "Sim":
            self.drone.takeoff()
        else:
            self.drone.safe_takeoff(5) # wait 5 seconds and make sure


    def stop(self):
        """
        Sets the DroneController waiting status to false and changes the drones stat
e to hovering
        """
        self.is_waiting = True
        if self.which_type == "Sim":
            self.drone.hover()


    def land(self):
        """
        Instructs the DroneController to land their drone.
        """
        if self.which_type != "Sim":
            self.is_active = False
            self.drone.safe_land()


    def get_altitude(self):
        """
        Returns a float representing the current altitude reading
        """
        return self.altitude_reading


    def set_altitude(self, altitude):
        """
        Overwrites the current altitude for the DroneControllers current location

        :param altitude: a float representation of the desired altitude for DroneCon
trollers current location
        """
        if self.which_type == "Sim":
            self.drone.set_altitude(altitude)


    def is_drone_active(self):
        """
```

```
        Returns a boolean expression that determines whether the DroneController is
active or not
        """
        return self.is_active


    def is_drone_moving(self):
        """
        Returns a boolean expression that determines whether the DroneController is
moving or not
        """
        return self.is_moving


    def is_drone_waiting(self):
        """
        Returns a boolean expression that determines whether the DroneController is
waiting or not
        """
        return self.is_waiting


    def is_route_complete(self):
        """
        Returns a boolean expression that determines whether the DroneController has
 completed its assigned route
        """
        return self.route_complete


    def get_point_travelling_to(self):
        """
        Returns a boolean expression that determines whether the DroneController has
 completed its assigned route
        """
        return self.point_moving_towards


    def is_drone_homeward_bound(self):
        """
        Returns a boolean expression that determines whether the DroneController is
moving towards home
        """
        return self.moving_home


    def get_heading(self):
        """
        Returns a boolean expression that determines whether the DroneController has
 completed its assigned route
        """
        azimuth = self.current_location.azimuthTo(self.next_point)
        return azimuth


    def get_current_location(self):
        """
        Returns the current location from the drone. Updates first to be sure it's a
n
        up-to-date reading
        """
        self.set_current_location()
        return self.current_location


    def set_current_location(self):
        """
        Set the current location for the DroneController. Gets the location from
        the drone its piloting.
        """
        if self.which_type == "Sim":
            self.current_location = self.drone.get_current_coordinate()
        else:
```

```
            curr_loc = self.drone.get_current_location()
            lat = curr_loc[0]
            lng = curr_loc[1]
            alt = curr_loc[2]

            location = QGeoCoordinate(lat, lng, alt)

            self.current_location = location


    def lower_altitude(self, n):
        """
        Depending on the type of drone we're piloting, subtract param n from the
        current altitude from our current coordinate.

        :param n: a float to subtract from our current altitude
        """
        if self.which_type == "Sim":
            if self.altitude_reading - n <= 10:
                self.drone.set_altitude(self.get_current_location().altitude() + n)
            else:
                self.drone.set_altitude(self.get_current_location().altitude() - n)
            self.set_current_location()


    def going_to(self):
        """
        Return the coordinate the DroneController is heading towards
        """
        return self.next_point


    def go_here(self, location):
        """
        Instructs the DroneController to move to param location.

        :param location: A QGeoCoordinate object containing the location the drone n
eeds
        to move towards.
        """
        if self.which_type == "Sim":
            if self.drone.ordered_to_wait() == True:
                return
            else:
                self.is_moving = True
                self.drone.set_temp_coordinate(location)
                self.drone.move()
                self.drone.swap()
                self.set_current_location()
                self.time_spent_travelling += 1.0
        else:
            # This code does not run as expected
            if self.drone.reached_location() == False:
                lat = location.latitude()
                lng = location.longitude()
                alt = location.altitude()
                self.drone.moveTo(lat, lng, alt, 'TO_TARGET')


    def hover(self):
        """
        Sets the moving status for the drone to false. Sets the drone to hover.
        """
        self.is_moving = False
        self.drone.hover()


    def resume(self):
        """
        Wakes the drone from its waiting state
        """
        self.is_waiting = False
        self.drone.resume()
```

```
    def get_time_spent_travelling(self):
        return self.time_spent_travelling


    def get_time_spent_travelling_towards_home(self):
        return self.time_spent_travelling_towards_home


    def reset_time_spent_travelling(self):
        self.time_spent_travelling = 0.0


    def get_current_speed(self):
        return self.speed


    def assign_flight_plan(self, fp):
        """
        Sets the flight plan assigned to the DroneController. The DroneController
        only knows the name of the flight plan. The AirTrafficController class
        will keep track of the DroneControllers place in the Flight Plan.

        :param fp: str, representing the name of the assigned flight plan
        """
        self.assigned_flight_plan = fp


    def get_home(self):
        """
        Returns the assigned home location for the DroneController

        :return home: A QGeoCoordinate containing the coordinates for the
        DroneControllers home location

        """
        return self.home


    def get_flight_plan_name(self):
        """
        Returns the name of the flight plan assigned to the DroneController

        :return: a str representing the name of the assigned flight plan
        """
        return self.assigned_flight_plan


    def speedup(self):
        """
        Increases the speed in kmph of the DroneController. Maxes at 60.0.
        """
        if self.speed < 60.0:
            self.speed += 10.0
        elif self.speed == 60.0:
            pass


    def slowdown(self):
        """
        Decreases the speed in kmph of the DroneController. Stops at 10.0
        """
        if self.speed > 10.0:
            self.speed -= 10.0
        elif self.speed == 10.0:
            pass


    def setup_video(self):
        """
        Setup the video functionality for the real drone.
        """
```

```
        if self.which_type != "Sim":
            stream_mode_parameters = ["low_latency", "high_reliability",
                                "high_reliability_low_framerate"]
            video_framerate_parameters = ["24_FPS", "25_FPS", "30_FPS"]

            # set up stream
            self.drone.set_video_stream_mode(stream_mode_parameters[2])
            self.drone.set_video_framerate(video_framerate_parameters[0])
            self.drone.start_video_stream()


    def close_feed_and_disconnect(self):
        """
        Ensures the drone has stopped streaming,
        and has successfully disconnected from the program
        before shutdown.
        """
        if self.which_type != "Sim":
            self.drone.stop_video_stream()
            self.drone.smart_sleep(2)
            print(self.drone.sensors.battery)
            self.drone.disconnect()


    def run_test(self):
        """
        Tests the connection to the real drone by issuing a
        take off and land command in succession.
        """
        if self.which_type != "Sim":
            self.drone.safe_takeoff(10)

            # set safe parameters
            self.drone.set_max_tilt(5)
            self.drone.set_max_vertical_speed(1)
            self.update_readings()
            self.drone.smart_sleep(5)

            self.drone.safe_land(10)


    def ready_to_move_again(self):
        """
        Determine if the real drone has reached its location
        """
        if self.which_type != "Sim":
            if self.drone.reached_location():
                return True
            else:
                return False


    def return_home(self):
        """
        Instructs the real drone to return to its home location
        """
        if self.which_type != "Sim":
            self.drone.return_home()
```

```python
# Module for FlightPlan objects
# Author: Ronan Donohue
# Date: Friday, Jan 25, 2019

__licence__ = "MIT"
__revision__ = "FlightPlan.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'


import numpy as np
from Loader import Loader

from PyQt5.QtPositioning import QGeoCoordinate

class FlightPlan:
    """
    This class acts as an ADT for coordinate information
    gotten from the user or recieved from the database.
    """

    def __init__(self, flight_plan_name):
        """
        Initialize the FlightPlan. If no name is passed as a paramter, a
        empty FlightPlan is initialized. Else, the corresponding
        FlightPlan info is pulled from the database into the object.

        :param flight_plan_name: a user generated description
        for a FlightPlan object. Used to retreive historic FlightPlan data.
        Defaults to none if parameter is omitted.
        """
        self.loader = Loader()
        self.flight_info = []
        self.flight_id = 0
        self.index = 0
        self.flight_plan_name = flight_plan_name
        self.create_flight_plan(flight_plan_name)
        self.flight_plan_length = len(self.flight_info)


    def create_flight_plan(self, flight_plan_name):
        """
        Create a FlightPlan object based off of the name parameter
        If name exists, get relevant data from the database
        If name is provided, but doesn't exist, create new flight plan
        with parameter name
        If parameter name is None, create a FlightPlan object with default name
        """
        if flight_plan_name is not None:
            if self.loader.is_flight_name_in_use(flight_plan_name):
                self.flight_info = self.loader.get_specific_flight_info(
                    flight_plan_name).copy()
                self.sort_flight_info_based_on_order()
                self.set_flight_plan_id()
            else:
                # Create new FlightPlan with name parameter
                self.loader.create_new_flight_plan(flight_plan_name)
                self.set_flight_plan_id()


    def sort_flight_info_based_on_order(self):
        """
        Sorts self.flight_info to reflect the desired order taken from
        the flight plan information gotten from the database.
        Sorts self.flight_info in place.
        An example would be:
        desired_order = '1, 2, 3, 4, 5'
        current_order = [2, 4, 5, 3, 1]
        self.sort_flight_info_based_on_order()
        current_order = [1, 2, 3, 4, 5]
        """
        correct_order = self.flight_info[0]['coordinate_order'].split(', ')
        correct_order = list(map(int, correct_order))
        sorted_flight_info = []
```

```
        len_flight_info = len(self.flight_info)

        for i in range(len_flight_info):
            for j in range(len_flight_info):
                if correct_order[i] == self.flight_info[j]['coordinate_id']:
                    sorted_flight_info.append(self.flight_info[j])
        self.flight_info = sorted_flight_info.copy()


    def set_flight_plan_id(self):
        """Set flight_id for object.
        Raises an exception if the FlightID isn't found"""
        f_id = self.loader.get_flight_id(self.flight_plan_name)
        if f_id != -1:
            self.flight_id = f_id[0]['flight_plan_id']
        else:
            raise ValueError("Flight ID not found, returned -1")


    def print_flight_plan_info(self):
        """Used for testing purposes, would never be needed otherwise"""
        ## TODO: remove this once no longer needed
        for item in self.flight_info:
            print(item)


    def get_entire_plan(self):
        return self.flight_info


    def increment_index(self):
        """Increments the internal index used for self.flight_info by 1"""
        self.index += 1


    def is_flight_plan_complete(self):
        """Returns True if we have iterated through self.flight_info.
        Raises False otherwise."""

        if self.index == len(self.flight_info):
            print("We're done")
            return True
        else:
            return False


    def get_coordinates(self):
        """returns a numpy array of floats representing the lat, long and alt
        coordinates from self.index location in self.flight_info"""
        if self.index == len(self.flight_info):
            print("We're equal")
            geo_coord = QGeoCoordinate()
            geo_coord.setLatitude(float(self.flight_info[self.index]['lat']))
            geo_coord.setLongitude(float(self.flight_info[self.index]['lng']))
            geo_coord.setAltitude(float(self.flight_info[self.index]['altitude']))
        else:
            geo_coord = QGeoCoordinate()
            geo_coord.setLatitude(float(self.flight_info[self.index]['lat']))
            geo_coord.setLongitude(float(self.flight_info[self.index]['lng']))
            geo_coord.setAltitude(float(self.flight_info[self.index]['altitude']))
        return geo_coord


    def get_altitude(self):
        """returns a float representation of the current altitude from
        self.index location in self.flight_info"""
        altitude = float(self.flight_info[self.index]['altitude'])
        return altitude


    def get_current_index(self):
        """Returns the current value (int) of self.index"""
        return self.index
```

```
    def get_index_of_coordinates(self, coordinates):
        """Returns an integer i representing the index location of the
        parameter coordinates if they are present in self.flight_info.
        Returns -1 if coordinates are not found/present.
        :param coordinates: a list of floats representing the x, y and z
        values to be searched for.
        :return: an integer representing the index location of
        parameter coordinates, or -1 if not present"""
        for i in range(len(self.flight_info)):
            if (coordinates[0] == float(self.flight_info[i]['lat'])
            and coordinates[1] == float(self.flight_info[i]['lng'])
            and coordinates[2] == float(self.flight_info[i]['altitude'])
            and coordinates[3] == float(self.flight_info[i]['bearing_to_next_point']
)):
                return i
        return -1


    def get_index_of_altitude(self, altitude):
        """Returns an integer i representing the index of parameter altitude
        if present within self.flight_info.
        Returns -1 if not present/found.
        :param altitude: a float representation of the altitude value to
        be searched for.
        :return: an integer representing the index location of
        parameter altitude, or -1 if not present"""
        for i in range(len(self.flight_info)):
            if altitude == float(self.flight_info[i]['altitude']):
                return i
        return -1


    def get_current_coordinate_order(self):
        """Returns a list representing the coordinate_order obtained from
        iterating though self.flight_info
        returns: a list of integers"""
        c_order = []
        for i in range(len(self.flight_info)):
            c_order.append(self.flight_info[i]['coordinate_id'])
        return c_order


    def set_current_coordinate_order(self):
        """Stores the current coordinate order for self.flight_info
        as a string in the database. The string is associated with
        the flight_plan name"""
        c_order = self.get_current_coordinate_order()
        str_order = ', '.join(str(x) for x in c_order)
        for i in range(len(self.flight_info)):
                self.flight_info[i]['coordinate_order'] = str_order
        self.loader.update_coordinate_order(self.flight_id,
                str_order)


    def append_new_coordinate(self, coordinates):
        """
        Appends new coordinate information (list)
        to the end of self.flight_info list.

        Once appended, the coordinates are passed to the database for
        storage and
        the current coordinate order is updated.
        :param coordinates: new coordinate list to be added to the
        flight_info list of dicts.

        parameter format should match the following:
        [x_coord, y_coord, z_coord, heading, altitude]
        """
        new_dict = {}
        new_dict['flight_plan_name'] = self.flight_plan_name
        new_dict['lat'] = coordinates[0]
```

```
        new_dict['lng'] = coordinates[1]
        new_dict['altitude'] = coordinates[2]
        new_dict['bearing_to_next_point'] = coordinates[3]

        self.save_coordinate(new_dict)
        new_dict['coordinate_id'] = self.loader.get_coordinate_id(
                new_dict)['coordinate_id']

        self.flight_info.append(new_dict)
        self.set_current_coordinate_order()


    def insert_new_coordinate(self, index, coordinates):
        """Inserts new coordinate information at location specified
        by parameter index within self.flight_info list.

        Once inserted, the coordinates are passed to the database for
        storage and
        the current coordinate order is updated.
        :param index: integer representation of location to add coordinates to.
        :param coordinates: new coordinates to be added to the flight_info list.
        parameter coordinates format should match the following:
        [lat, lng, altitude, bearing_to_next_point]
        """
        new_dict = {}
        new_dict['flight_plan_name'] = self.flight_info[self.index][
                'flight_plan_name']
        new_dict['lat'] = coordinates[0]
        new_dict['lng'] = coordinates[1]
        new_dict['altitude'] = coordinates[2]
        new_dict['bearing_to_next_point'] = coordinates[3]

        self.save_coordinate(new_dict)
        new_dict['coordinate_id'] = self.loader.get_coordinate_id(
                new_dict)['coordinate_id']
        self.flight_info.insert(index, new_dict)
        self.set_current_coordinate_order()


    def remove_coordinates(self, coordinates):
        """
        Remove the coordinates specified by parameter coordinates if
        they are present within self.flight_info list.
        If they are not, raise ValueError() and inform user.
        Once found, the coordinates are removed from the database also
        and the coordinate_order is updated to reflect this.
        :param coordinates: an array/np array
        format [lat, lng, alt, bearing_to_next_point]
        """
        inx = self.get_index_of_coordinates(coordinates)
        if inx == -1:
            raise ValueError("Coordinate not found in FlightPlan, returned " +
                    str(inx))
        else:
            coord_id = self.flight_info[inx]['coordinate_id']
            self.loader.delete_coordinate_data_in_db(coord_id)
            self.update_coordinate_order(coord_id)
            self.flight_info.pop(inx)


    def update_coordinate_order(self, coord_id):
        """
        This function updates the coordinate_order in the database with
        the current coordinate_order via the Loader object.
        """
        current_order = self.flight_info[0]['coordinate_order'].split(', ')
        current_order = list(map(int, current_order))
        db_string = ', '.join(str(i) for i in current_order if i != coord_id)
        self.loader.update_coordinate_order(
                self.flight_info[self.index]['flight_plan_id'], db_string)


    def update_coordinates(self, index, coordinates):
```

```
        """
        Update coordinates at parameter index with
        the new coordinates from parameter coordinates.
        :param index: the location of the coordinates to
        be updated.
        :param coordinates: the new coordinates to be
        copied over.
        """
        keys_needed = ['coordinate_id', 'lat', 'lng', 'altitude',
                'bearing_to_next_point']

        subdict_of_flight_info = new_dict = dict(
                (k, self.flight_info[index][k]) for k in keys_needed if k in self.fl
ight_info[index])
        self.loader.update_coordinates(subdict_of_flight_info)



    def save_coordinate(self, new_coordinates):
        """
        This function writes the new coordinate information to the
        database via the Loader object.
        """
        new_coordinates['flight_plan_id'] = self.flight_id
        self.loader.insert_new_coordinate(new_coordinates)


    def get_all_flight_plan_info(self):
        returned_rows = self.loader.get_all_info()
        return returned_rows


    def get_all_flight_plan_names(self):
        all_names = self.loader.get_flight_plan_names()
        return all_names


    def get_flight_plan_length(self):
        return len(self.flight_info)


    def reset_flight_plan(self):
        self.index = 0


    def decrement_index(self):
        self.index -= 1
```

```python
# Module for Loader objects
# Author: Ronan Donohue
# Date: Friday, 25 Jan, 2019

__license__ = "MIT"
__revision__ = "Loader.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'


import pymysql.cursors

class Loader:
    """
    This class is responsible for all CRUD FlightPlan
    actions performed on the database.
    """

    def __init__(self):
        """Upon initalization, connect to database"""
        self.establish_connection()


    def establish_connection(self):
        """Connects to database.
        Autocommit is enabled for all queries executed.
        """
        self.db = pymysql.connect(host = "localhost",
                                  user = "root",
                                  password = "root",
                                  #db='datcs',
                                  autocommit=True,
                                  cursorclass=pymysql.cursors.DictCursor)
        with self.db.cursor() as cursor:
            query_string = """
            create database if not exists datcs;
            """
            cursor.execute(query_string)
            query_string = """
            use datcs;"""
            cursor.execute(query_string)
            query_string = """
            CREATE TABLE IF NOT EXISTS flight_plan(
            flight_plan_id INT AUTO_INCREMENT,
            flight_plan_name VARCHAR(255) NOT NULL,
            coordinate_order VARCHAR(255),
            created_at DATETIME,
            delete_flag TINYINT,
            PRIMARY KEY (flight_plan_id)
            ) ENGINE=INNODB;
            """
            cursor.execute(query_string)
            query_string = """
            CREATE TABLE IF NOT EXISTS gps_coordinates(
            coordinate_id INT AUTO_INCREMENT,
            flight_plan_id INT,
            FOREIGN KEY (flight_plan_id) REFERENCES flight_plan(flight_plan_id),
            lat DOUBLE,
            lng DOUBLE,
            altitude DECIMAL,
            delete_flag TINYINT,
            PRIMARY KEY (coordinate_id)
            ) ENGINE=INNODB;
            """
            cursor.execute(query_string)


    def insert_new_coordinate(self, coordinate_info):
        """Adds new coordinate info into coordinates table in database

        :param coordinate_info: a dict containing the information to be saved"""
        with self.db.cursor() as cursor:
            query_string = """insert
            into gps_coordinates
```

```
            (flight_plan_id, lat, lng, altitude, bearing_to_next_point,
            delete_flag)
            values
            (%s, %s, %s, %s, %s, 0);"""
            cursor.execute(query_string, (coordinate_info['flight_plan_id'],
                coordinate_info['lat'],
                coordinate_info['lng'],
                coordinate_info['altitude'],
                coordinate_info['bearing_to_next_point']))


    def get_coordinate_id(self, coordinate_info):
        """Returns the coordinate_id associated with parameter dict supplied from
        the database.

        :param coordinate_info: dict of info used to obtain the coordinate_id
        from the database.
        :return: the coordinate_id (int) for the given coordinates"""
        with self.db.cursor() as cursor:
            query_string = """
            select coordinate_id
            from gps_coordinates
            where
            lat = %s
            and lng = %s
            and altitude = %s
            and bearing_to_next_point = %s
            and delete_flag = 0;"""
            cursor.execute(query_string, (coordinate_info['lat'],
                coordinate_info['lng'],
                coordinate_info['altitude'],
                coordinate_info['bearing_to_next_point']))
            result = cursor.fetchone()
        return result


    def delete_flight_data_in_db(self, flight_plan_id):
        """Sets the corresponing delete_flag to 1
        for parameter flight_plan_id"""
        with self.db.cursor() as cursor:
            query_string = """update
            flight_plan
            set delete_flag = 1
            where flight_plan_id = %s
            and delete_flag = 0;"""
            cursor.execute(query_string, (flight_plan_id));


    def delete_coordinate_data_in_db(self, coord_id):
        """
        Sets the delete_flag to 1 for the corresponding coord_id
        """
        with self.db.cursor() as cursor:
            query_string = """update
            gps_coordinates
            set delete_flag = 1
            where coordinate_id = %s;"""
            cursor.execute(query_string, (coord_id))


    def update_coordinate_order(self, flight_plan_id, coord_order):
        """
        Sets the new coordinate_order for a flight plan.
        :param1 flight_plan_id: the name of a given flight plan
        :param2 coordinate_order: the new coordinate_order string to be updated
        """
        with self.db.cursor() as cursor:
            query_string = """update
            flight_plan
            set coordinate_order = %s
            where flight_plan_id = %s;"""
            cursor.execute(query_string, (coord_order, flight_plan_id))
```

```
    def get_specific_flight_info(self, flight_plan_name):
        """
        Retrieve flight_plan information associated with parameter
        name
        :return: a list of dicts with all coordinate information for the
        parameter flight_plan_name
        """
        with self.db.cursor() as cursor:
            query_string = """select
            *
            from flight_plan fp inner join gps_coordinates c
            on fp.flight_plan_id = c.flight_plan_id
            where fp.flight_plan_name = %s
            and (fp.delete_flag = 0 and c.delete_flag = 0);
            """
            row_count = cursor.execute(query_string, (flight_plan_name))
            if row_count > 0:
                result = cursor.fetchall()
            else:
                result = None
        return result


    def is_flight_name_in_use(self, name):
        """
        Returns True if parameter name is already in use,
        else returns False
        :param name: the name of a proposed FlightPlan
        """
        with self.db.cursor() as cursor:
            query_string = """select
            * from flight_plan where
            flight_plan_name = %s;"""
            row_count = cursor.execute(query_string, (name))
            if row_count > 0:
                return True
            else:
                return False


    def create_new_flight_plan(self, flight_plan_name):
        """
        Creates a new FlightPlan instance in the database
        with parameter flight_plan_name as the name.
        :param flight_plan_name: the name for the new FlightPlan
        """
        with self.db.cursor() as cursor:
            query_string = """insert
            into flight_plan
            (flight_plan_name, coordinate_order, delete_flag)
            values
            (%s, '', 0);"""
            cursor.execute(query_string, (flight_plan_name))


    def get_flight_id(self, flight_plan_name):
        """
        Return the flight_id for parameter flight_plan_name
        Returns -1 if not present.
        :param flight_plan_name: the name to search the database for
        """
        with self.db.cursor() as cursor:
            query_string = """
            select flight_plan_id
            from flight_plan where
            flight_plan_name = %s and delete_flag = 0;"""
            row_count = cursor.execute(query_string, (flight_plan_name))
            if row_count > 0:
                result = cursor.fetchall()
            else:
                result = -1
        return result
```

```
    def update_coordinates(self, coordinates):
        """
        Overwrite the coordinates at parameter coordinate_id with
        parameter dict coordiantes values.
        :param coordinates: a dict of new coordinate data
        """
        with self.db.cursor() as cursor:
            query_string = """update
            gps_coordinates
            set lat = %s
            , lng = %s
            , altitude = %s
            , bearing_to_next_point = %s
            where coordinate_id = %s;"""
            cursor.execute(query_string, (coordinates['lat'],
                coordinates['lng'],
                coordinates['altitude'],
                coordinates['bearing_to_next_point'],
                coordinates['coordinate_id']))


    def get_all_info(self):
        """
        Returns all info for all FlightPlan data
        """
        with self.db.cursor() as cursor:
            query_string = """select *
            from flight_plan fp
            join gps_coordinates gps
            on fp.flight_plan_id
            = gps.flight_plan_id
            where (fp.delete_flag = 0
            and gps.delete_flag = 0);"""
            row_count = cursor.execute(query_string)
            if row_count > 0:
                result = cursor.fetchall()
            else:
                result = None
        return result


    def get_flight_plan_names(self):
        """
        Returns all the names for FlightPlans in the db
        """
        with self.db.cursor() as cursor:
            query_string = """select
            flight_plan_name
            from flight_plan
            where delete_flag = 0;"""
            row_count = cursor.execute(query_string)
            if row_count > 0:
                result = cursor.fetchall()
            else:
                result = None
            return result
```

```
# Author: Ronan Donohue
# Number: c00208501

__license__ = "MIT"
__revision__ = "SimulatedDrone.py 25/01/2019 Ronan Donohue"
__docformat__ = 'reStructuredText'

from PyQt5.QtPositioning import QGeoCoordinate
from PyQt5.QtCore import (QObject, QTimer, pyqtSignal,
        pyqtSlot, pyqtProperty, QTime)

import time

class SimulatedDrone(QObject):
    """
    A simple class meant to represent certain aspects of a drone in Python.

    :param home_coordinate: A QGeoCoordinate object with coordinates for home
    """

    def __init__(self, home_coordinate, parent = None):
        """
        Intiializes the SimulatedDrone object
        """
        super().__init__(parent)
        self.home = home_coordinate
        self.next_coordinate = QGeoCoordinate(0.0, 0.0, 0.0)
        self.temp_coordinate = QGeoCoordinate(0.0, 0.0, 0.0)
        self.current_coordinate = home_coordinate
        self.speed = 22.00
        self.waiting = False
        self.battery_level = 100
        self.current_state = "landed"
        self.possible_states = tuple(["landed", "taking off",
                         "flying", "emergency", "landing", "hovering"])


    def set_next_coordinate(self, next_coordinate):
        """
        Sets the next coordinate the SimulatedDrone is to travel towards

        :param next_coordinate: a QGeoCoordinate containing the next location the dr
one must visit
        """
        self.next_coordinate = next_coordinate


    def set_temp_coordinate(self, new_current_coordinate):
        """
        Set the temporary coordinate used to mimic travel time

        :param new_current_coordinate: a QGeoCoordinate containing the temporary coo
rdinate
        """
        self.temp_coordinate = new_current_coordinate


    def get_current_coordinate(self):
        """
        Returns a QGeoCoordinate object containing our current location
        """
        return self.current_coordinate


    def set_altitude(self, altitude):
        """
        Sets the altitude for the SImulatedDrones current location

        :param altitude: a floating-point representation of the desired altitude
        for the SimulatedDrones current altitude
        """
        self.current_coordinate.setAltitude(altitude)
```

```
    def takeoff(self):
        """
        Change the current_state of the SimulatedDrone to
        taking off, then hovering
        """
        self.current_state = self.possible_states[
            self.possible_states.index("taking off")
        ]
        self.current_state = self.possible_states[
            self.possible_states.index("hovering")
        ]


    def stop(self):
        """
        Change the current state of the SimulatedDrone to
        hoveirng, and set the speed to 0 and change the
        waiting status to True
        """
        self.current_state = self.possible_states[
            self.possible_states.index("hovering")
        ]
        self.speed = 0
        self.waiting = True


    def resume(self):
        """
        Change the waiting status of the SimulatedDrone to False,
        we are ready to continue
        """
        self.waiting = False
        self.current_state = self.possible_states[
            self.possible_states.index("flying")
        ]


    def hover(self):
        """
        Change the current state of the SimulatedDrone to
        hoveirng, and set the speed to 0
        """
        self.current_state = self.possible_states[
            self.possible_states.index("hovering")
        ]
        self.speed = 0


    def ordered_to_wait(self):
        """
        Returns the waiting status of the SimulatedDrone
        """
        return self.waiting


    def move(self):
        """
        Changes the current_state of the SimuatedDrone to flying
        """
        self.current_state = self.possible_states[
            self.possible_states.index("flying")
        ]


    def swap(self):
        """
        Overwrite current_coordinate with temp.coordinate
        """
        self.current_coordinate = self.temp_coordinate
```