BIRMINGHAM CITY University

Technical Report: **DRONE DETECTION**
Authors: **DAVID COTTRELL**
Date: **08/05/20**
Wordcount: **1486**
Pagecount: **20**
Confidential: **NO**

Contents

Executive Summary

The following will describe an algorithm created to help prevent drone interference within airports.

The airport's main airport building and runway will be represented as two separate lists of points.

A series of coordinates will be randomized at runtime along with a randomly selected Radar Cross Section from the provided drones_list file. These coordinates will be used to represent the coordinates of drones flying over or near the airport. The make and model of each Drone are determined by performing a Binary Search on the sorted list within drones_list, using the randomly selected Radar Cross Section as a target.

The Matplotlib library was used to help visualize the output of the algorithm on a 2D plot. The two airport polygons are plotted along with all the randomized coordinates. If a pair of coordinates are within a polygon, their point on the plot is coloured red, otherwise it will be coloured blue.

I was able to break the problem into 5 stages: sorting the list within drones_list; filling the **Drones** data structure; determining if a set of coordinates are within a polygon; searching for the make and model and displaying the out data to a graph.

The implementation of the algorithm is successful as it worked as expected. The time efficiency is shown in Figure 11 and could likely be improved by using a compiled language.

Introduction

Drones have become widely available for public use; unforeseen issues have recently been discovered. On 20th December 2018, “around 1,000 flights were disrupted” costing Gatwick airport “£1.4m” (Topham, 2019) in losses due to drones flying over the airport, making the use of algorithms such as this one important to airports.

Theory

Initially, the data required to represent the polygons are stored as NumPy arrays. As the provided contains function requires data to be stored in this format. These NumPy arrays are then casted to array-lists, as plotting them using Matplotlib, the separate list components (X and Y data) of each array would need to be separated using Python’s zip function – which requires the lists to be of type array-list.

The main data structure used throughout the program is an array-list named **Drones.** This structure holds all relevant information required for a drone within my program as separate list items. This information consists of X and Y coordinates (stored as a Tuple as they will not need to be changed), Radar Cross Section (inferred as an int), make and model (stored as an array-list as these will be determined later) and whether the coordinates are within one of the polygons (stored as a Boolean).

At the beginning of the algorithm, each list item consists of randomly generated X and Y coordinates; a randomly selected Radar Cross Section from the drones_list file and a default False value for whether the coordinates are within a polygon.

The Boolean can be used later by the plotting function to display points within the polygons differently than ones outside.

Keeping all the data for the drones together in a large list allows for more simple code that is easier to read and understand and helps prevent data getting modified unexpectedly.

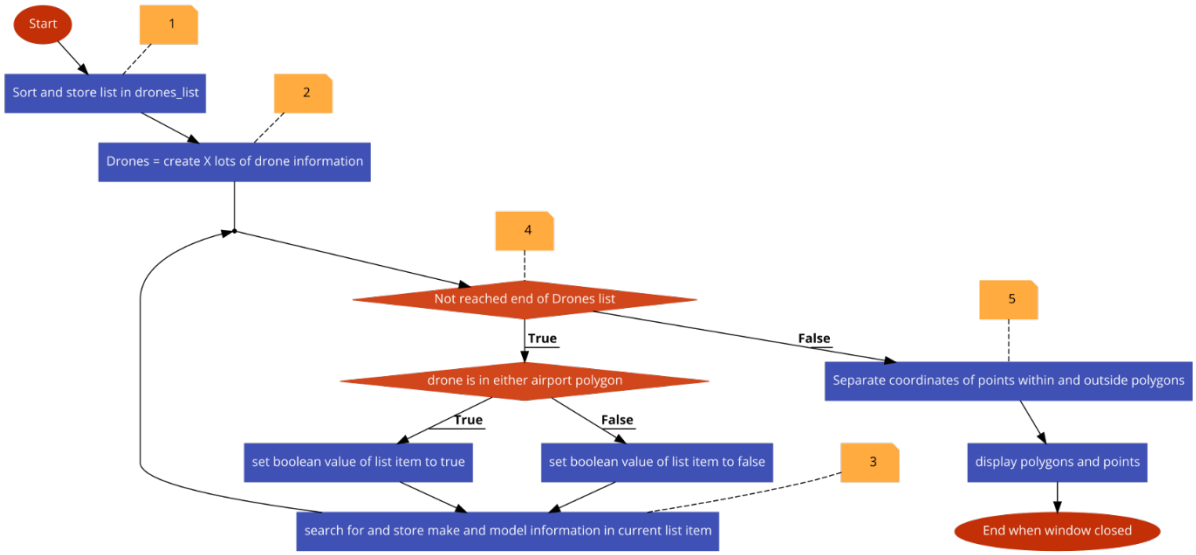

Figure 1 – Process flow

Figure 1 shows a high-level flow diagram of the process the algorithm will take and can be broken down into 5 main components.

1. Sorting the drones list

Python's built-in sorted function was used to sort the list within drones_list, as is required for the Binary search algorithm.

Sanatan (2020) stated the sorted function will "implement the Tim Sort algorithm". Thus, a time complexity of "$O(n * \log(n))$" (GeeksforGeeks, 2020).

2. Filling the **Drones** data structure

This fills the Drones list with random coordinates, a randomly chosen Radar Cross Section from the provided drones list and placeholder information for data to be determined later. This loop has a time complexity of $O(n)$.

3. Searching for the make and model

The make and model associated with a given Radar Cross Section need to be determined for each list item within **Drones**.

To do this a binary search algorithm was used, alerted from (GeeksforGeeks, 2020) and Pythons built in Sorted function to sort the provided drones_list. It is important for this to be time efficient, as it will be repeated many times, once per **Drones** list item.

The Binary search algorithm has a time complexity of "$O(\log(n))$" (GeeksforGeeks, 2020) but requires the list to be sorted before it can be used.

4. Collecting information from points inside and outside the polygons

This will iterate over each element within the **Drones** data structure, meaning it will iterate m times, where m is the number of elements within **Drones,** so has a time complexity of $O(m)$. It will check if the coordinates of the current drone are within either polygon, using the provided contains function

(Appendix C). The contains function will iterate x times, where x is the length of the polygon provided to the function. This function has a time complexity of $O(x^{2.5})$.

The main loop of this function contains the binary search algorithm, resulting in a combined time complexity of $O(x^{2.5} * m * \log(n))$.

5. Display data

This prepares and plots the polygon and drone coordinate data to a graph, allowing you to hover over your mouse curser over a drone to display information about that drone.

It will require the **Drones** data structure to be iterated over once, along with many $O(1)$ operations, giving this function a time complexity of $O(n)$, where n is the number of list elements within **Drones**.

Implementation

The implementation of the 5 main components detailed in the theory section are given below.

1. Filling the **Drones** data structure

This function uses Python's random library to generate the random coordinates assigned to **x** and **y** and a random index used to retrieve a Radar Cross Section from the drones_list. This information is stored in a list and repeated **num** times until the loop is finished and the list is returned.

```
def createDrones(num):
    DroneInfo = []
    for _ in range(0, num):
        x = random.randrange(0, 1200) #Random X coordinate between 0 and 1200
        y = random.randrange(0, 700) #Random Y coordinate between 0 and 700
        RCS = L[random.randrange(0, len(L))][5] #Get a random radar cross section from the drone list
        DroneInfo.append( [ (x, y), RCS, ["", ""], False ] )
    return DroneInfo
```

Figure 2 - CreateDrones

2. Sorting the drones list

This sorts the list within drones_list **L** using the 5th index of each list item using Python's sorted function as detailed above.

```
sortedDroneList = sorted(L, key=lambda x: x[5])
```

Figure 3 – sorted

3. Searching for the make and model

The below implements a binary search, using the provided Radar Cross Section (RCS) as a target value. If the RCS is found it will a return an array-list with the make and model, otherwise it will raise an error and execution will stop.

```
def getMakeModel(sortedDroneList, l, r, RCS):
    while l <= r:
        mid = l + (r - l)//2
        if sortedDroneList[mid][5] == RCS:
            return [sortedDroneList[mid][0], sortedDroneList[mid][1]]
        elif sortedDroneList[mid][5] < RCS:
            l = mid + 1
        else:
            r = mid - 1
    raise ValueError("Error: RCS not found")
```

Figure 4 – getMakeModel

4. Collecting information from points inside and outside the polygons

The coordinates of each element in the **Drones** list are checked if they are within either polygon and their corresponding Boolean value is set accordingly. The Binary search is then used to find the make and model of the current drone.

```
def findDrones(sortedDroneList, Drones):
    for Drone in Drones:
        # coordinates inside a polygon
        if contains(AirpoirtMain, np.array(Drone[0])) or contains(AirpoirtRunway, np.array(Drone[0])):
            Drone[3] = True
        # coordinates outside a polygon
        else:
            Drone[3] = False
        Drone[2] = getMakeModel(sortedDroneList, 0, len(sortedDroneList)-1, Drone[1])
```

Figure 5 – findDrones

5. Display data

Once the positions of all the drone coordinates have been determined, they can be plotted on a grid along with the polygons. This is shown below using Matplotlib.

```
def showPlot(AirpoirtMain, AirpoirtRunway, Drones):
    # Prepare polygon data for matplotlib
    AirpoirtMain = AirpoirtMain.tolist()
    AirpoirtMain.append(AirpoirtMain[0])
    mainX, mainY = zip(*AirpoirtMain)

    AirpoirtRunway = AirpoirtRunway.tolist()
    AirpoirtRunway.append(AirpoirtRunway[0])
    RunwayX, RunwayY = zip(*AirpoirtRunway)

    # graph setup
    figure = plt.figure()
    axes = figure.add_subplot()

    insideCoords = [[], []]
    outsideCoords = [[], []]

    for Drone in Drones:
        if(Drone[3] == True):
            insideCoords[0].append(Drone[0][0])
            insideCoords[1].append(Drone[0][1])
        else:
            outsideCoords[0].append(Drone[0][0])
            outsideCoords[1].append(Drone[0][1])

    scatter1 = axes.scatter(insideCoords[0], insideCoords[1], color='red', s=10) # Add scatter graph for drones
found to be within either airport polygon
```

```
    scatter2 = axes.scatter(outsideCoords[0], outsideCoords[1], color='blue', s=10) # Add scatter graph for drones not in either airport polygon

    axes.plot(mainX, mainY, RunwayX, RunwayY) # Draw polygons

    annotation = axes.annotate("", xy=(0,0), xytext=(-100,20), textcoords="offset points", color=(1, 1, 1), bbox=dict(fc=(0, 0, 0, 1), ec='none', pad=0.5, boxstyle="round"), arrowprops=dict(arrowstyle="->")) # Annotation style
    annotation.set_visible(False) # Hide annotation

    figure.canvas.mpl_connect("motion_notify_event", lambda event: hover(event, figure, axes, annotation, scatter1, scatter2) ) # Listen for hover events
    plt.show() # Show plot
```

Figure 6 – showPlot

Testing

To ensure the algorithm was working correctly, I entered 4 sets of drone coordinates and the polygon points into an online graphing tool (Desmos, 2020) (Figure 7).

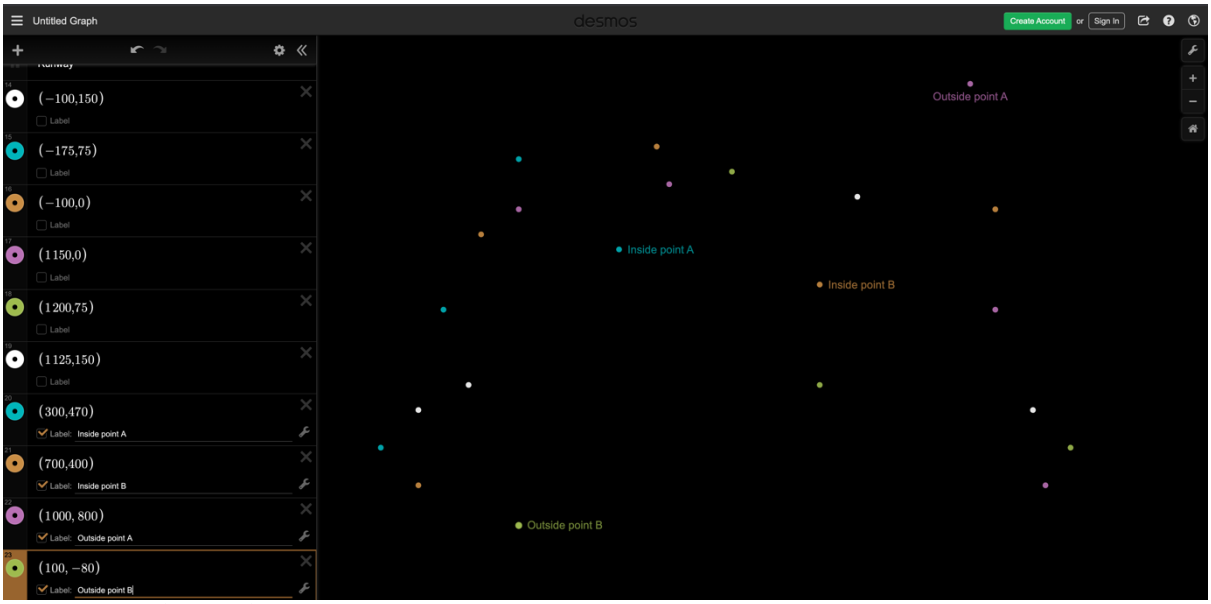

Figure 7 – Graphing Tool

This allowed me to enter 2 sets of points that I knew would be either inside or outside the polygons into my algorithm (Figure 8).

```
Drones = [
    [(300, 470), 4941000, ["", ""], False], #'ProFlight', 'c_maxi'
    [(700, 400), 9945600, ["", ""], False], #'Yuneec', 'YYrZrm5Y1_maxi'
    [(1000, 800), 9628500, ["", ""], False], # 'Yuneec', '33X5aa'
    [(100, -80), 1579662, ["", ""], False] #'DJI', 'XaX5rppac_maxi'
]
```

Figure 8 – Drone data entered manually

Figure 9 shows the algorithm raised no errors.

```
176  findDrones(sortedDroneList, Drones)
177
178  assert Drones[0][2][0] == 'ProFlight'
179  assert Drones[0][2][1] == 'c_maxi'
180  assert Drones[0][3] == True
181
182  assert Drones[1][2][0] == 'Yuneec'
183  assert Drones[1][2][1] == 'YYrZrm5Y1_maxi'
184  assert Drones[1][3] == True
185
186  assert Drones[2][2][0] == 'Yuneec'
187  assert Drones[2][2][1] == '33X5aa'
188  assert Drones[2][3] == False
189
190  assert Drones[3][2][0] == 'DJI'
191  assert Drones[3][2][1] == 'XaX5rppac_maxi'
192  assert Drones[3][3] == False
193

PROBLEMS   TERMINAL

~ » /usr/local/bin/python3 "/Users/dave/Documents/Programming/Drone Finder/algorithm.py"
~ »
```

Figure 9 – correctness testing

Figure 10 shows correct plotting.

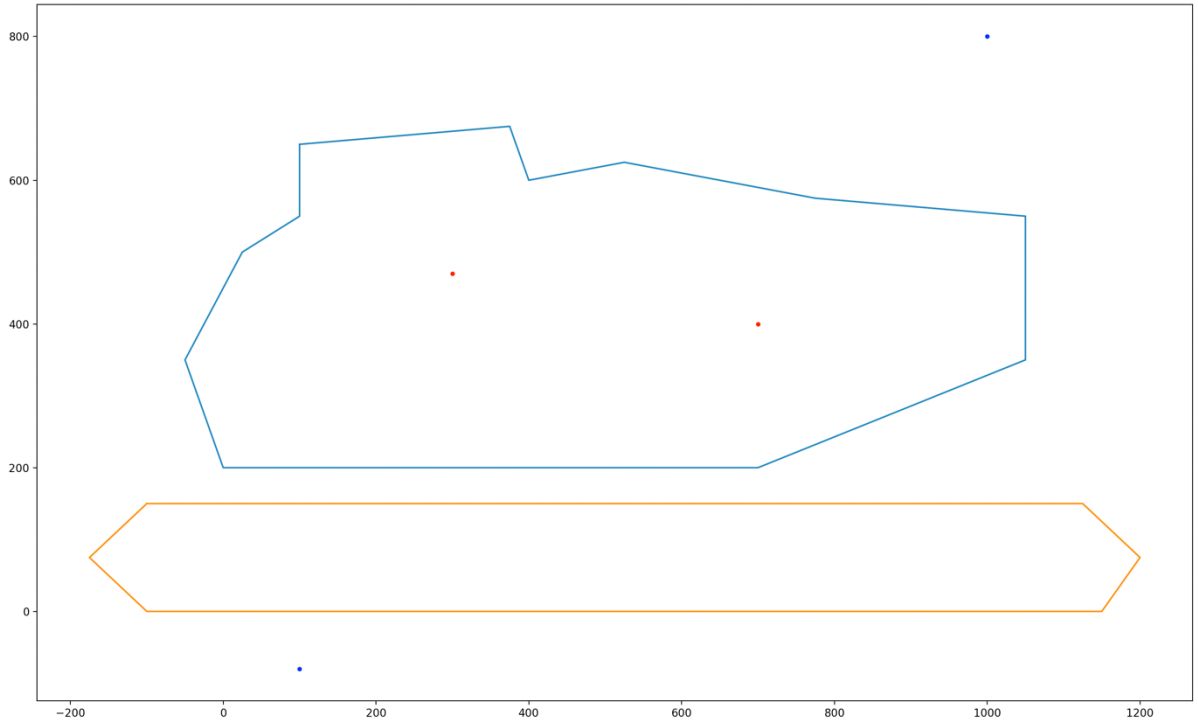

Figure 10 – correctness testing plot

Dynamic analysis

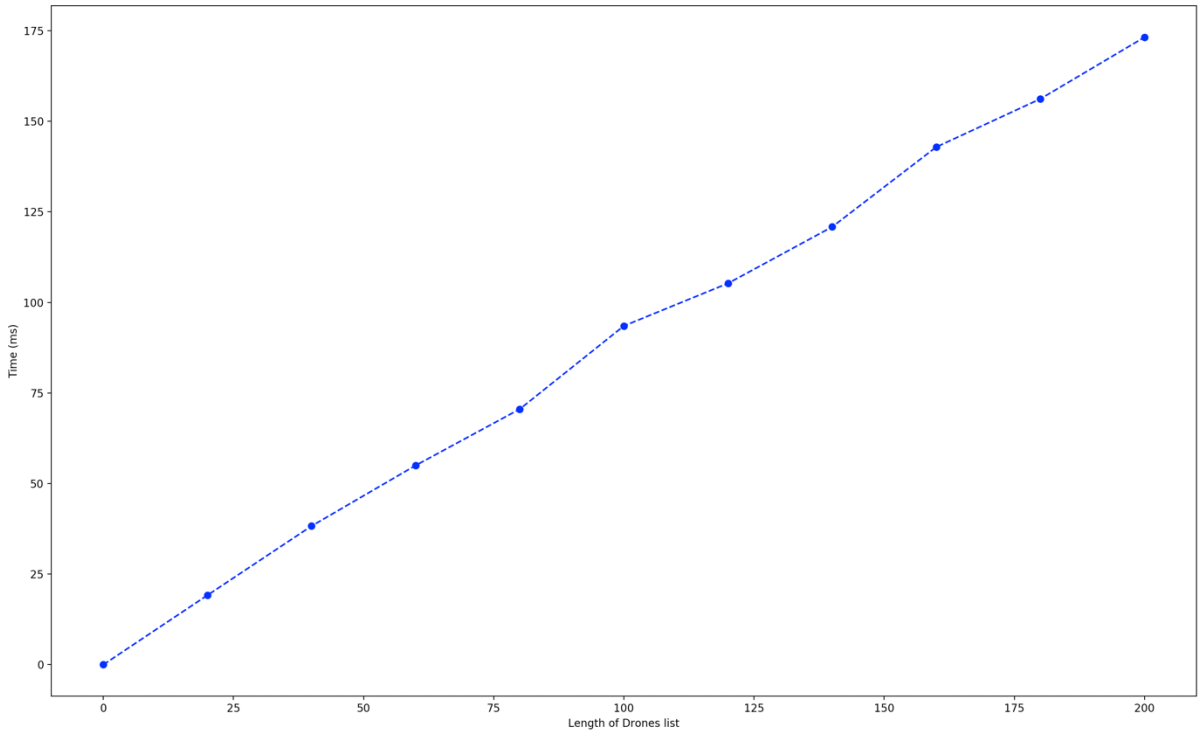

Figure 11 - Time plot

Figure 11 shows the average time taken by the algorithm to process an increasing number of elements within the **Drones** list and was the output of the timeAnalysis function (Appendix A). The average time was found by performing the same calculation 500 times for each size of the **Drones** list.

Conclusions

The algorithm worked as expected. It runs sufficiently quickly for an example setting, as realistically a large number of drones would not be flying over an airport at one time. Figure 11 shows the algorithm runs in linear time, meaning that doubling the input data doubles the compute time. The program also allows the user to visualize the output of the program intuitively.

However, this type of software has no tolerance for error, as a mistake or fault in the program could lead to the loss of potentially hundreds of lives. This is an example of a safety critical system, and in the real world should be written in a more stable language other than Python. Some of the features the alternative language should include are being low-level; strictly typed; compiled and without garbage collection.

Compiled languages are generally much faster, which is important as detecting drones as fast as possible is essential in this setting. Type checking would make the system more secure as type inference could lead to errors. Garbage collection has the possibility of producing spontaneous errors, so having complete control over memory would be more secure. As stated by ConcernedOfTunbridgeWells (2017)

> While manual memory management code must be carefully checked to avoid errors, it allows a degree of control over application response times that is not available with languages that depend on garbage collection.

Some of Python's features include it being dynamically typed; garbage collected and interpreted, meaning it is a poor choice for safety critical situations. Some of the key questions to be answered when choosing a language for safety critical software are:-

- "Are the means of data typing strong enough to prevent misuse of variables?
- Are there facilities in the language to guard against running out of memory at runtime? (e.g. to prevent stack or heap overflow.)
- If the software detects a malfunction at runtime, do mechanisms exist to facilitate recovery?" (Cullyer, Goodenough and Wichmann, 1991).

References

ConcernedOfTunbridgeWells. (2017). *Which languages are used for safety-critical software?.* Available through: https://stackoverflow.com/a/243573 [Accessed 5 May 2020].

Cullyer, J., Goodenough, J., Wichmann, A., (1991). *The choice of computer languages for use in safety-critical systems.* Available through: https://pdfs.semanticscholar.org/827d/5c1c1b4ec83d92e957784baf314f2a2ddf0a.pdf [Accessed 5 May 2020].

Desmos. (2020). Available through: https://www.desmos.com/calculator [Accessed 5 May 2020].

GeeksforGeeks. (2020). *Complexity Analysis of Binary Search.* Available through: https://www.geeksforgeeks.org/complexity-analysis-of-binary-search/ [Accessed 5 May 2020].

GeeksforGeeks. (2020). *TimSort.* Available through: https://www.geeksforgeeks.org/timsort/ [Accessed 5 May 2020].

Sanatan, M. (2020). *Sorting Algorithms in Python*. Available through: https://stackabuse.com/sorting-algorithms-in-python/ [Accessed 5 May 2020].

Topham, G. (2019). *Gatwick drone disruption cost airport just £1.4m*. Available through: https://www.theguardian.com/uk-news/2019/jun/18/gatwick-drone-disruption-cost-airport-just-14m [Accessed 5 May 2020].

Appendix A

```
import numpy as np
import matplotlib.pyplot as plt
import random
from time import perf_counter

from drones_list import L
from contains import contains

def createDrones(num):
    DroneInfo = []
    for _ in range(0, num):
        x = random.randrange(0, 1200) #Random X coordinate between 0 and 1200
        y = random.randrange(0, 700) #Random Y coordinate between 0 and 700
        RCS = L[random.randrange(0, len(L))][5] #Get a random radar cross section from the drone list
        DroneInfo.append( [ (x, y), RCS, ["", ""], False ] )
    return DroneInfo

def getMakeModel(sortedDroneList, l, r, RCS):
    while l <= r:
        mid = l + (r - l)//2
        if sortedDroneList[mid][5] == RCS:
            return [sortedDroneList[mid][0], sortedDroneList[mid][1]]
        elif sortedDroneList[mid][5] < RCS:
            l = mid + 1
        else:
            r = mid - 1
    raise ValueError("Error: RCS not found")

# Determine if drone is within polygons and add its coordinates and name to the appropriate arrays
def findDrones(sortedDroneList, Drones):
    for Drone in Drones:
        # coordinates inside a polygon
        if contains(AirpoirtMain, np.array(Drone[0])) or contains(AirpoirtRunway, np.array(Drone[0])):
            Drone[3] = True
        # coordinates outside a polygon
        else:
            Drone[3] = False
        Drone[2] = getMakeModel(sortedDroneList, 0, len(sortedDroneList)-1, Drone[1])


def showAnnotation(ind, graph, list, annotation):
    pos = graph.get_offsets()[ind["ind"][0]]
    annotation.xy = pos
    if list == "trueList":
        text = "Within airport\nDrone make: {}\nDrone model: {}\nRadar Cross Section: {}\nCoordinates: ({},
{})".format(
            Drones[int(ind["ind"])][2][0],
            Drones[int(ind["ind"])][2][1],
            Drones[int(ind["ind"])][1],
            Drones[int(ind["ind"])][0][0],
```

```
            Drones[int(ind["ind"])][0][1]
        )
    else:
        text = "Outside airport\nDrone make: {}\nDrone model: {}\nRadar Cross Section: {}\nCoordinates: ({},
{})".format(
            Drones[int(ind["ind"])][2][0],
            Drones[int(ind["ind"])][2][1],
            Drones[int(ind["ind"])][1],
            Drones[int(ind["ind"])][0][0],
            Drones[int(ind["ind"])][0][1]
        )

    annotation.set_text(text)


def hover(event, figure, axes, annotation, scatter1, scatter2):
    # Hover event over points within scatter1
    if scatter1.contains(event)[0]:
        contains, index = scatter1.contains(event)
        # If points are very close to eachother, it combines both indexes into an array
        if contains and len(index["ind"]) == 1:
            showAnnotation(index, scatter1, "trueList", annotation)
            annotation.set_visible(True)

    # Hover event over points within scatter2
    elif scatter2.contains(event)[0]:
        contains, index = scatter2.contains(event)
        if contains and len(index["ind"]) == 1:
            showAnnotation(index, scatter2, "falseList", annotation)
            annotation.set_visible(True)

    # Hide annotation when mouse left point
    else:
        annotation.set_visible(False)

    # Show changes on canvas
    figure.canvas.draw_idle()


def showPlot(AirpoirtMain, AirpoirtRunway, Drones):
    # Prepare polygon data for matplotlib
    AirpoirtMain = AirpoirtMain.tolist()
    AirpoirtMain.append(AirpoirtMain[0])
    mainX, mainY = zip(*AirpoirtMain)

    AirpoirtRunway = AirpoirtRunway.tolist()
    AirpoirtRunway.append(AirpoirtRunway[0])
    RunwayX, RunwayY = zip(*AirpoirtRunway)

    # graph setup
    figure = plt.figure()
```

```
    axes = figure.add_subplot()

    insideCoords = [[], []]
    outsideCoords = [[], []]

    for Drone in Drones:
        if(Drone[3] == True):
            insideCoords[0].append(Drone[0][0])
            insideCoords[1].append(Drone[0][1])
        else:
            outsideCoords[0].append(Drone[0][0])
            outsideCoords[1].append(Drone[0][1])

    scatter1 = axes.scatter(insideCoords[0], insideCoords[1], color='red', s=10) # Add scatter graph for drones found to be within either airport polygon
    scatter2 = axes.scatter(outsideCoords[0], outsideCoords[1], color='blue', s=10) # Add scatter graph for drones not in either airport polygon

    axes.plot(mainX, mainY, RunwayX, RunwayY) # Draw polygons

    annotation = axes.annotate("", xy=(0,0), xytext=(-100,20), textcoords="offset points", color=(1, 1, 1), bbox=dict(fc=(0, 0, 0, 1), ec='none', pad=0.5, boxstyle="round"), arrowprops=dict(arrowstyle="->")) # Annotation style
    annotation.set_visible(False) # Hide annotation

    figure.canvas.mpl_connect("motion_notify_event", lambda event: hover(event, figure, axes, annotation, scatter1, scatter2) ) # Listen for hover events
    plt.show() # Show plot


def timeAnalysis(sortedDroneList):
    timeList = []
    lengthList = []
    averageTimeList = []

    #0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200
    for i in range(0, 11):
        for _ in range(0, 500):
            Drones = createDrones(i*20)

            start_time = perf_counter()
            findDrones(sortedDroneList, Drones)
            end_time = perf_counter()

            time = 1000 * (end_time - start_time)
            timeList.append(time)

        averageTimeList.append(sum(timeList) / len(timeList))
        lengthList.append(i*20)
        print("Average for", i*20, "drones is", sum(timeList) / len(timeList))
        timeList = []
```

```
    plt.plot(lengthList, averageTimeList, '--bo')
    plt.xlabel('Length of Drones list')
    plt.ylabel('Time (ms)')
    plt.show()

AirpoirtMain = np.array((
    [100, 650], [375, 675], [400, 600], [525, 625], [775, 575], [1050, 550], [1050, 350], [700, 200], [0, 200], [-
50, 350], [25, 500], [100, 550]
)) # Polygon for aiport

AirpoirtRunway = np.array((
    [-100, 150], [-175, 75], [-100, 0], [1150, 0], [1200, 75], [1125, 150]
)) # Polygon for runway


sortedDroneList = sorted(L, key=lambda x: x[5])

#timeAnalysis(sortedDroneList)

Drones = createDrones(30) # Create 30 drones

findDrones(sortedDroneList, Drones)

showPlot(AirpoirtMain, AirpoirtRunway, Drones)
```

Appendix B

Contains.py (unmodified)

```
def contains(P,R):

    # import numpy as np
    # Polygon P = np.array(([80,150],[100,40],[140,170],[250,60],[320,150],[250,130],[110,250]))
    # Point R = np.array([200,140])
    import numpy as np
    intercepts = 0

    r_origin = R
    r_end = np.array([np.max(P[:,0])+30,np.max(P[:,1])+30])
    #print(r_origin)
    #print(r_end)
    r_direction = r_end - r_origin



    for i in range(len(P)):
        u_origin = P[i,:]
        u_direction = P[(i+1)%len(P),:] - u_origin
        Z = np.array([u_direction,-r_direction])
        #print(Z.T)
        intercept_len = np.linalg.pinv(Z.T).dot(r_origin - u_origin)
        #print(u_origin)
        #print(u_direction)
        #print(intercept_len)
        #print(sum((intercept_len>0) & (intercept_len<1)))
        if (((intercept_len[0]>0) & (intercept_len[0]<1)) and ((intercept_len[1]>0) & (intercept_len[1]<1))) == 1:
            intercepts += 1
        #print(intercepts)

    #print(intercepts)
    if intercepts%2 == 1:
        print('R is inside polygon P')
    else:
        print('R is outside polygon P')
```

Appendix C

Contains.py (Modified)

```
def contains(P,R):

    import numpy as np
    intercepts = 0

    r_origin = R
    r_end = np.array([np.max(P[:,0])+30,np.max(P[:,1])+30])
    r_direction = r_end - r_origin

    for i in range(len(P)):
        u_origin = P[i,:]
        u_direction = P[(i+1)%len(P),:] - u_origin
        Z = np.array([u_direction,-r_direction])
        intercept_len = np.linalg.pinv(Z.T).dot(r_origin - u_origin)
        if (((intercept_len[0]>0) & (intercept_len[0]<1)) and ((intercept_len[1]>0) & (intercept_len[1]<1))) == 1:
            intercepts += 1

    if intercepts%2 == 1:
        return True
    else:
        return False
```

Appendix D

Drones_list.py can be found here:
https://moodle.bcu.ac.uk/pluginfile.php/7024795/mod_resource/content/1/drones_list.py