Fuel-o-tron 1.0

Fuel Management Software

Documentation

Benjamin Murray DT265

Student Number: C10728329

Table of Contents

1.1 Program Overview

Author: Benjamin Murray

Student Number: C10728329

Data Created: 28th April 2016

Due Date: 1st May 2016

Course: OO Programming

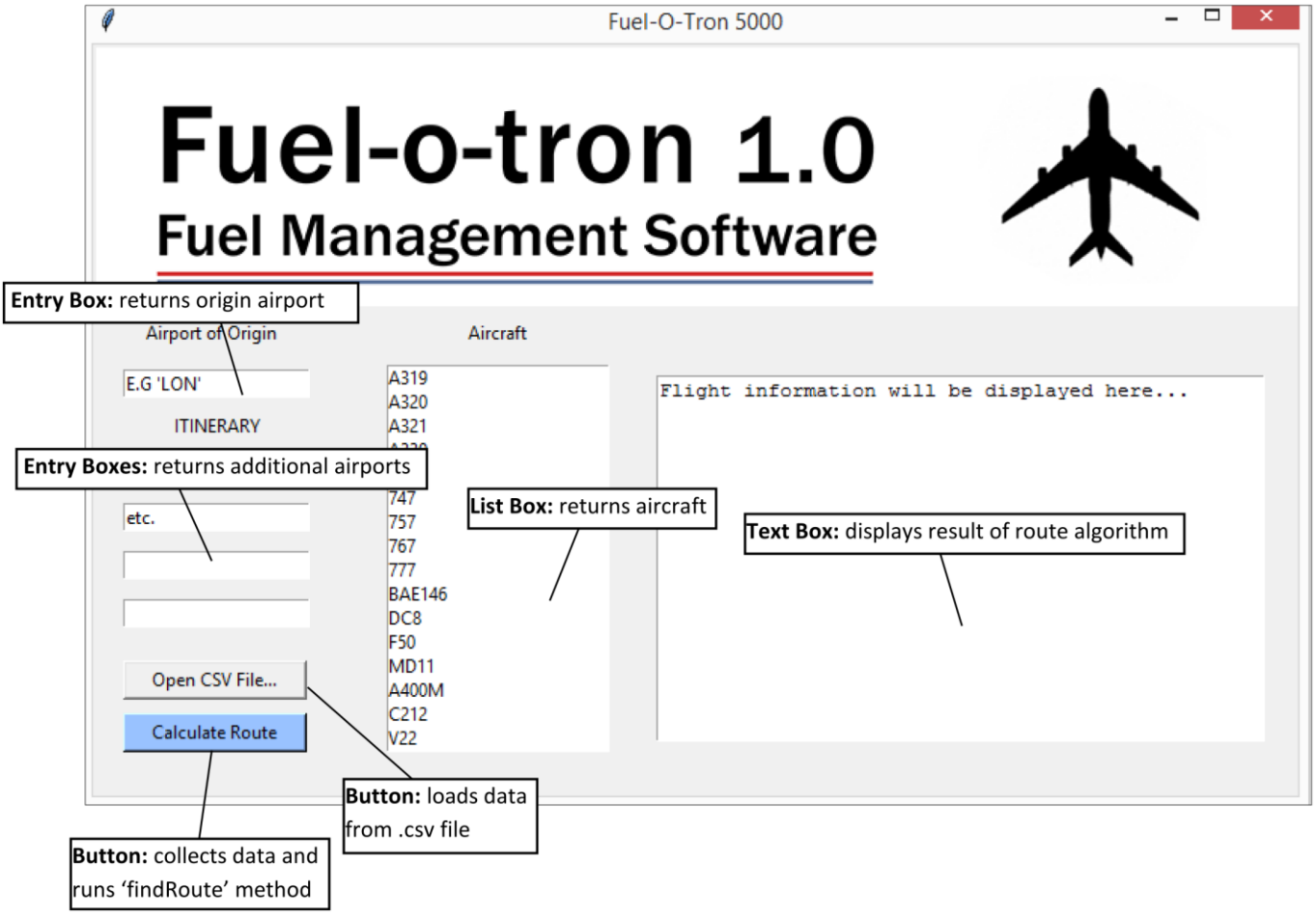
Lecturer: Dr. Aneel Rahim

Filename: fuelmanagement.py

Description: Fuel-o-tron is a fuel management software written in version Python 3.5. The program takes as input several .csv files containing data on airports, aircraft, currency and exchange rates – as well as a user-inputted itinerary – and attempts to calculate the most economic route based on this information.

1.2 Program Interface

The program's graphical user interface (GUI) is built using the tkinter module. The GUI consists of 9 individual widgets an image file organised using grid geometry. The widgets allow the user to input string data into the interface which is integrated into the program using .get() functions on the user pressing the 'calculate Route' button. The interface also allows for the display of the data generated by the program via a text box. Alternatively, the data may be written to an empty .csv file. The program is terminated by clicking the 'X' button located on right-hand corner of the program window.



1.3 Program Execution

The program is executed as follows:

1. The user inputs an airport of origin

2. The user inputs several additional airports that are to be included in the itinerary.

3. The user selects an aircraft from the aircraft list-box (if not, the program will default to first aircraft on the list)

4. The user presses the ‘calculate route’ button

5. The route information, total distance, fuel consumption and currencies utilised of the inputted itinerary are displayed in the text-box.

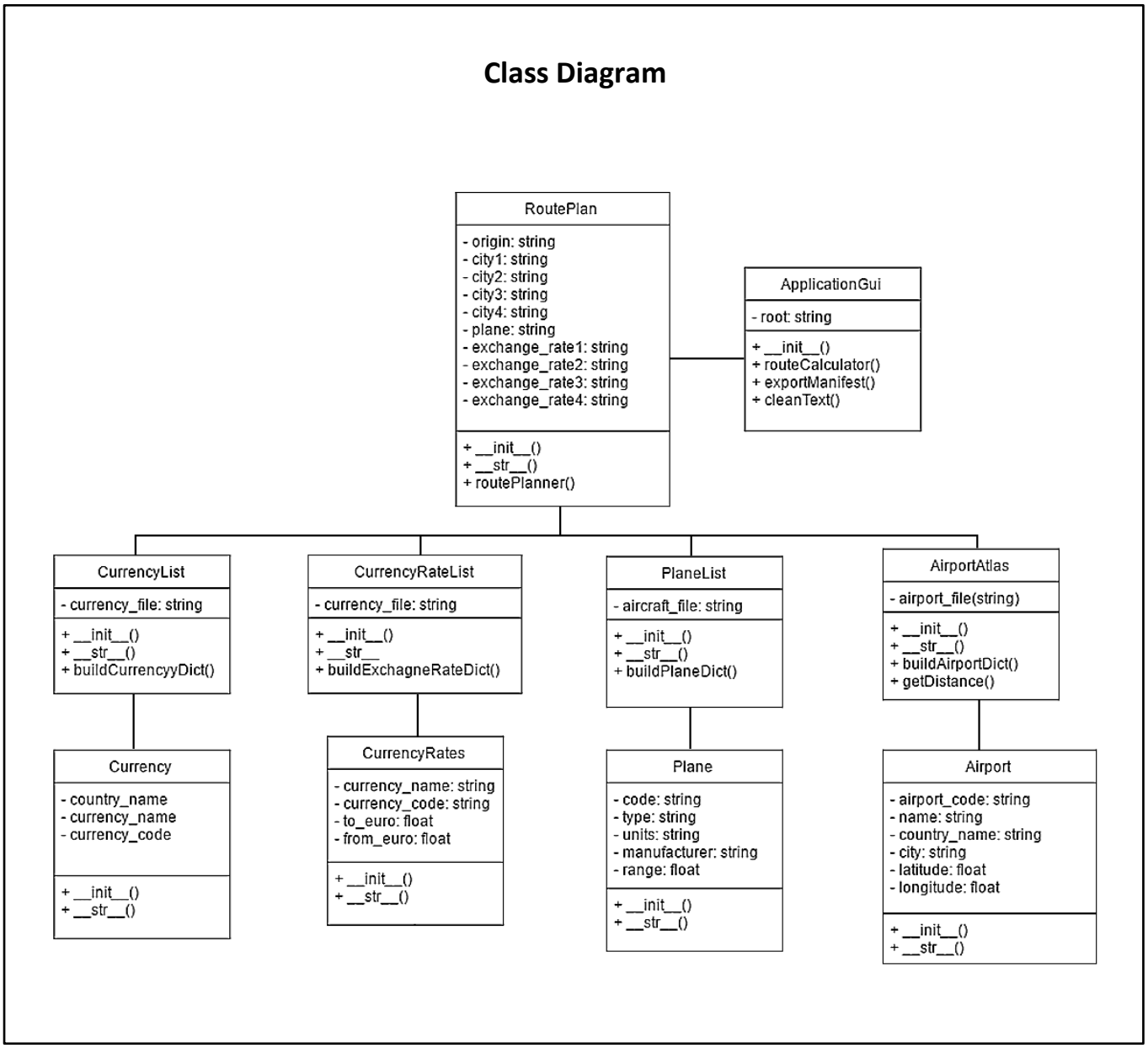
Alternative Step 4: The user presses the ‘write to .CSV file’ button and the program writes the flight data to an empty .csv file.

Alternate Step 5a: System informs user that the flight data is incorrect.

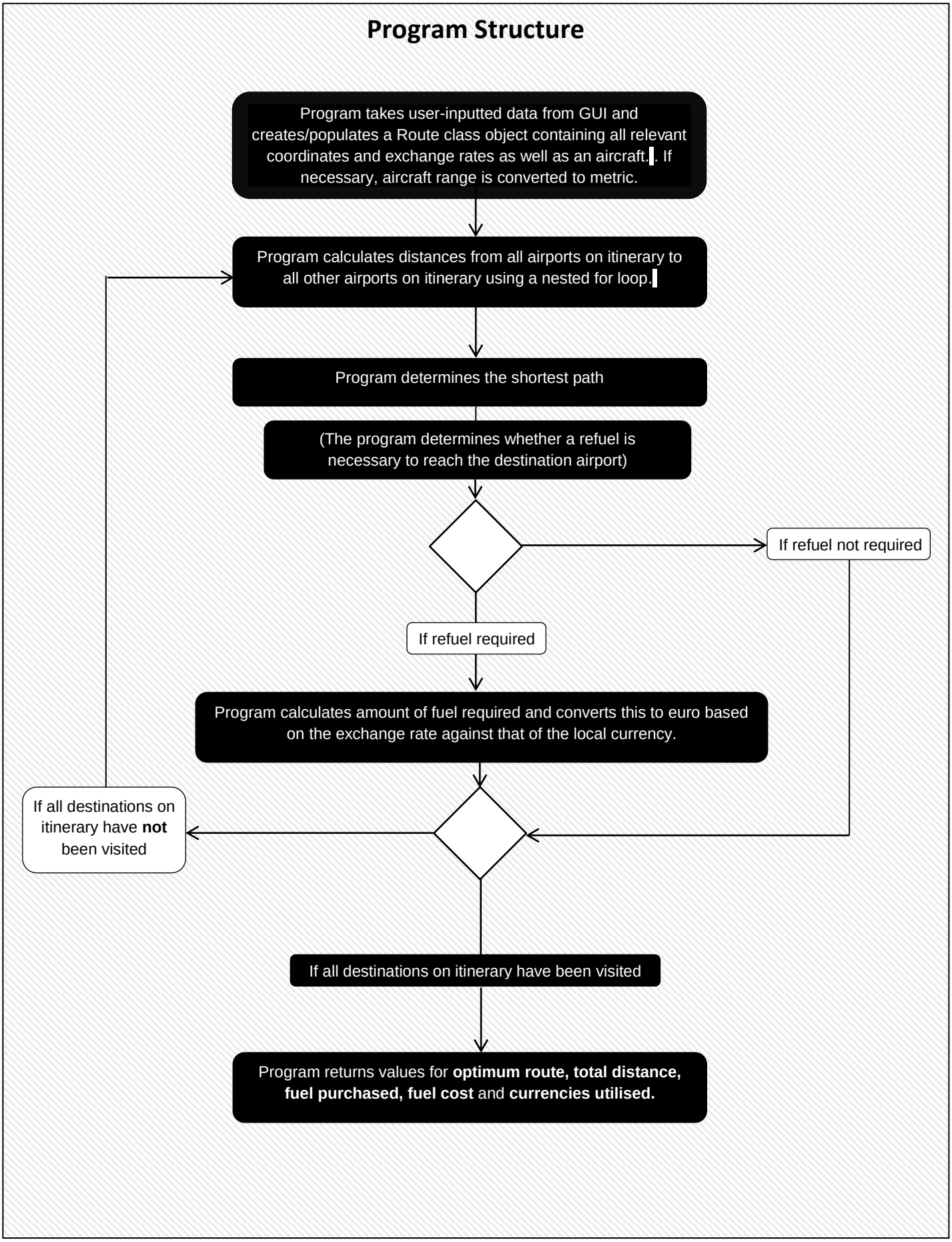
Alternate Step 5b: System informs user that one of the journeys on the itinerary exceeds the flight range of the selected aircraft.

2.1 Program Structure

The program relies on three main classes, namely **Airport, Plane**, **Currency** and **CurrencyRates.** Utilising data from four .csv files, objects from these classes are populated into dictionaries within corresponding classes called **AirportAtlas, PlaneList, CurrencyRateList** and **CurrencyList.** The latter classes contain most of the required methods for determining the program output. To consolidate the above classes and abstract the inner workings of the program from the GUI class, a class called **RoutePlan** creates an object containing all of the attributes necessary to access the other classes.



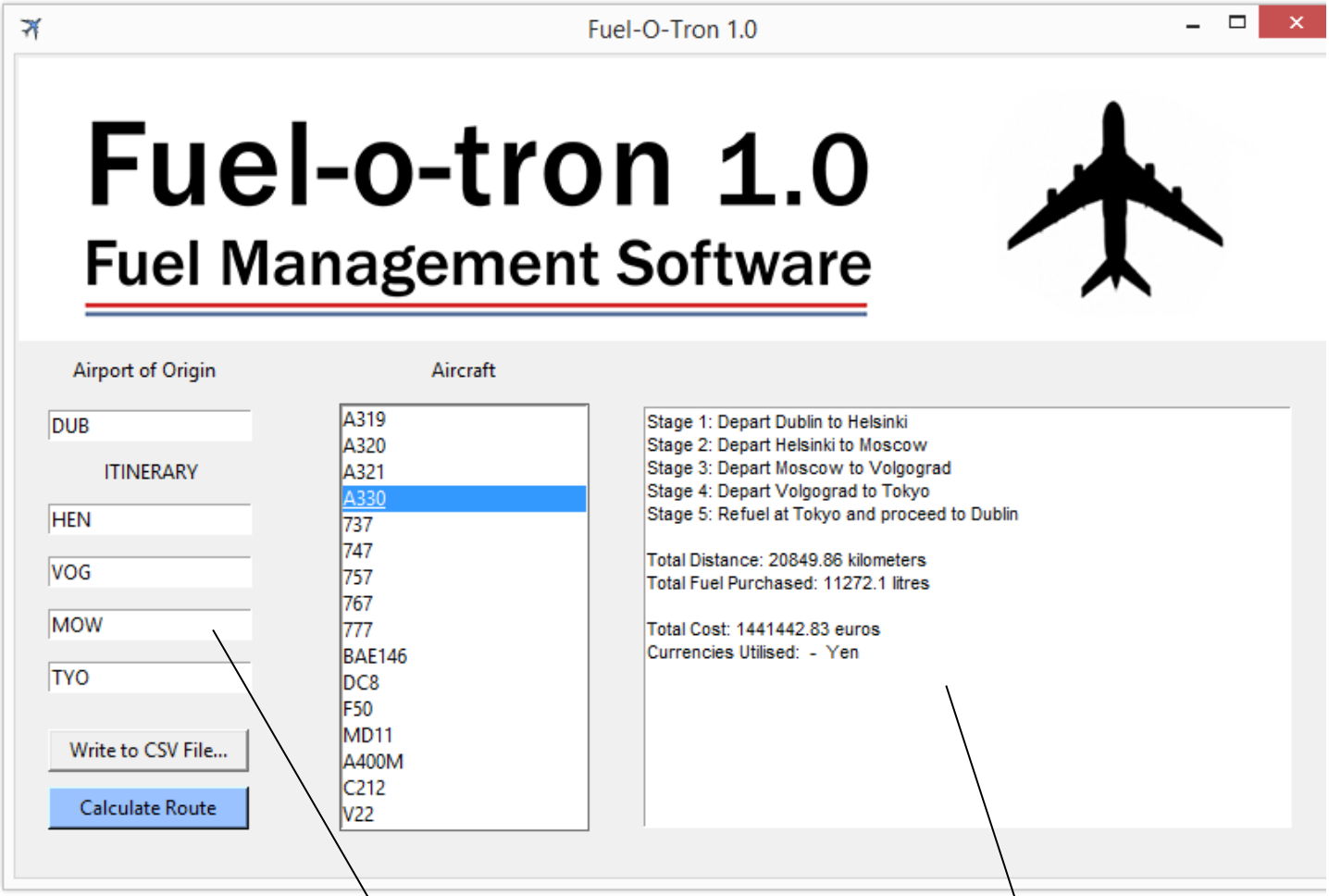
NOTE: All classes and associated methods are documented in detail in the appendix of this document.


Program Structure
Program takes user-inputted data from GUI and creates/populates a Route class object containing all relevant coordinates and exchange rates as well as an aircraft. If necessary, aircraft range is converted to metric.
Program calculates distances from all airports on itinerary to all other airports on itinerary using a nested for loop.
Program determines the shortest path
(The program determines whether a refuel is necessary to reach the destination airport)
If refuel not required
If refuel required
Program calculates amount of fuel required and converts this to euro based on the exchange rate against that of the local currency.
If all destinations on itinerary have **not** been visited
If all destinations on itinerary have been visited
Program returns values for **optimum route, total distance, fuel purchased, fuel cost** and **currencies utilised.**

2.4 Examples

Example 1:

<u>Normal Execution</u>



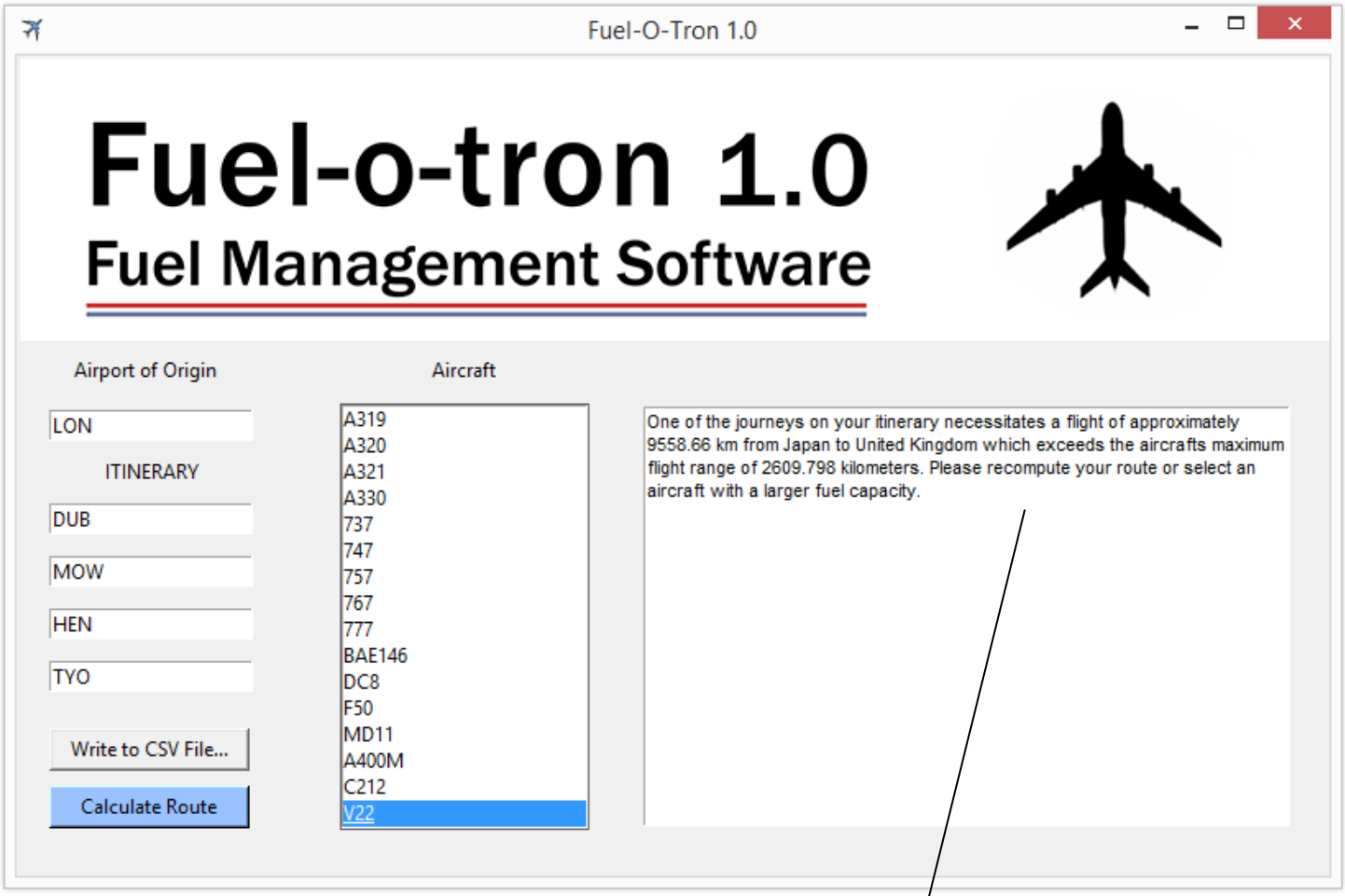
User has entered a valid

System displays

1) optimum itinerary
2) refuelling points
3) total distance
4) fuel purchased
5) currencies utilised

Example 2:

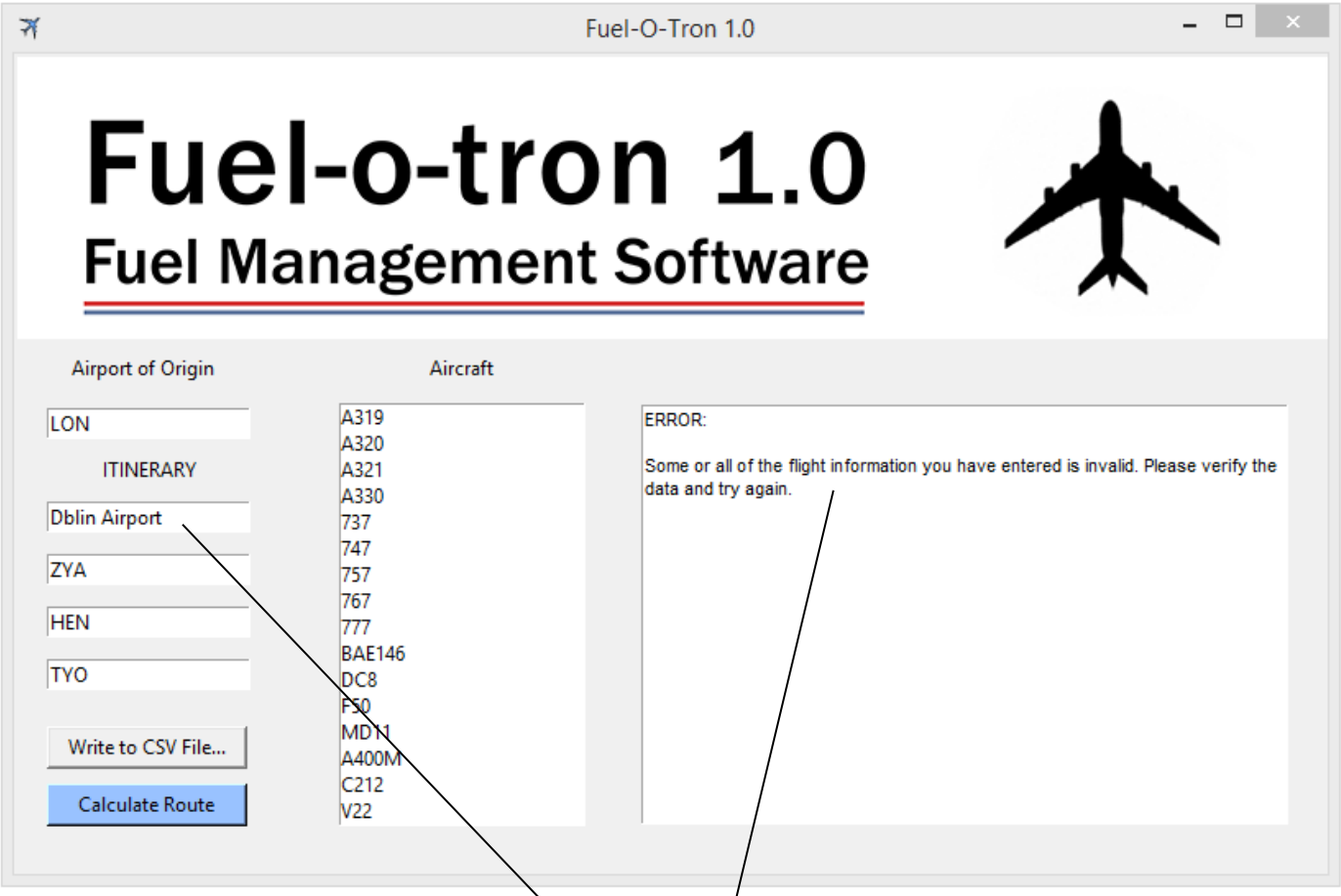
Alternate Execution #1



The user selects an itinerary which necessitates a journey which exceeds the maximum flight range of the selected aircraft.

Example 3:

<u>Alternate Execution #3</u>



The user enters invalid data. The system prompts the user to verify the data and try again.

2.3 Testing

Unit Testing

In order to ensure that the program executed as desired, it was necessary to develop a test suite. This test suite allowed for the chosen functions to be tested outside of the main program files which rendered troubleshooting much more effective. The following functions were tested utilising Python's in-built 'unittest' module. These functions were chosen for testing on the basis that the core functionality of the program was reliant on them:

- AirportAtlas.getDistance()
- CurrencyList.getCurrency()
- CurrencyRateList.getExchangeRate()
- PlaneList.getPlane()
- Route.routePlanner()

User Survey

In addition to testing the backend of the program, the graphical user interface underwent additional testing by means of a user survey. During this process, several users unfamiliar with the program, and possessing no programming experience, attempted to correctly execute the program with without assistance. On completion, the users were asked to present feedback in the form of a short survey. Utilising this information, the graphical user interface was able to undergo significant improvement in terms of its overall design and ease of use.

APPENDIX

Name:	**Airport**
Attributes:	Airport_code (string), name (string), country_name(string), city(string), latitude (float), longitude (float).
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation

Name:	**Plane**
Attributes:	Code(string), type(string), units(string), manufacturer(string), range(float)
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation

Name:	**Airport Atlas**
Attributes:	airport_file(string)
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation ========== buildAirportDict() **Description:** builds a dictionary of airport objects **Parameters:** filename(string) **Returns:** airports(dictionary) ========== getDistance() **Description:** calculates distance between two points **Parameters:** code1 (tuple), code2 (tuple) **Returns:** distance (float)

Name:	**PlaneList**
Attributes:	aircraft_file(string)
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation ========== __buildPlaneDict__ **Description:** builds a dictionary of plane objects **Parameters:** filename(string) **Returns:** planes (dictionary)

Name:	**CurrencyRates**
Attributes:	Currency_name(string), currency_code(string), to_euro(float), from_euro(float)
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation

Name:	**CurrencyList**
Attributes:	currency_file(string)
Methods:	__init__ **Description:** constructor ========== __str__ **Description:** string representation ========== __buildCurrencyDict__ **Description:** builds a dictionary of currency objects **Parameters:** filename(string) **Returns:** currencies (dictionary)

Name:	**ApplicationGui**
Attributes:	root
Methods:	__init__ **Description:** constructor ========== routeCalculattor() **Description:** calculates optimum route **Returns:** none