# Computer Science Coursework

## Active Route Finder

---

Name: Hamza Maqsood

Candidate Number: 9157

Centre Number: 10504

School: Bow School

# Contents Of Coursework Documentation

# Analysis

## Initial Problem and Identification

The initial problem this software is trying to solve is a program that's set towards the need for an industrial or individual modelling tool to actively recommend routes to reduce travel times towards private sector pilots or hobbyists that don't have access to corporation specific software such as highly recognised navigation apps or want a different software to meet their bespoke needs and provide different features, based on global or domestic trips.
A significant group of pilots are private and therefore do not have access to advanced route systems such as ones incorporated with companies aircraft or do not have access to software that meets their specific requirements as they may be a one size fits all approach, so a software has to be created to provide a service in the market for which there is a gap.
Current pilots may use base software included with their aircraft, therefore the computer software must retain key features of existing software that are necessary for flights whilst also providing a bespoke and more customised interface which can improve navigation.
This problem is suited to a computational approach due to the access of openstreetmap and its connected modules to allow the user to plot down coordinates onto the map through markers.
The geographical area would span domestically within the UK and have a possibility of allowing machine learning to predict or recommend a next best node along the route.

The initial simulation would also be incorporated into a machine learning algorithm to make sure it is becoming more accurate with every simulation it runs.
Furthermore data analytics and statistics will be integrated alongside this software in order to provide features such as estimated travel time, distance travelled, fuel consumption and costs which could be used personally, commercially or by firms in order to monitor usage.

One final addition to the software will include a booking system which can be used by private flyers in order to create a consistent schedule they can follow, this will help to improve efficiency and prevent confusion in their services.

## Uses of Computational Methods

**Thinking Abstractly:**
Abstract thinking is the use of reducing unnecessary information from a problem to leave only important information that is actually used to solve the initial problem and this will allow for easier problem solving.
Within the Active Route Finder abstraction could be used during the analysis of the map to remove unnecessary information such as colours, places not mentioned in the initial query etc, in order to leave the waypoints, paths and destination and departure locations so the algorithm can effectively and efficiently produce a model for how the user would move towards the destination in the most efficient manner possible.
Some representational abstractions we can add now is removing the complex functionality of the active route finder from the users view, so any command line outputs for development will not be shown to the user as well as hiding the processes of validations from the user and only showing them the outputs

**Thinking Concurrently:**

Due to computers being able to handle concurrent or parallel processing along with multitasking, this can be used to an advantage in the program to analyse and record patterns received from an input, which will therefore make the program more efficient by reducing the rendering time and allowing for a quicker response time to the user.
Furthermore the machine learning algorithm could use this to analyse different parts of the imputed area at the same time to reduce processing time, however some negative aspects about parallel processing would be that the computer can only handle so much data on each cores in the first instance so parallel processing would further reduce the amount that each core can handle at any given time.

**Thinking Procedurally:**

For the active route finder multiple components will be executed procedurally and in order, the order of these components and their steps are crucial as they later on contribute to the whole of the softwares execution.
Using some procedural abstraction we can identify that in order, the user opens the software, they then enter input data into the fields and then carry on to use the main functionality of the application in order to create their desired route.
The procedure interface for the software will be kept quite simple as from a development point of view I will still be able to identify how different modules interact with each other and how they work individually.
Through the top-down design approach and problem decomposition the active route finder will be able to be broken down into its main components in the design stage in order to ensure the proper and robust development of the software.

## Stakeholders

The main target stakeholders for the Active Route Finder will be private pilots and hobbyists that do not have access to advanced routing apps that are supplied to employees in airline companies and such, this software will allow the pilots to have access to a high quality software that will provide features key to the commercial airliner ones but without a one size fits all approach and also incorporate extra features in order to improve the quality of service. This will include features such as fuel usage, estimated travel time and distance to the destination ; furthermore the routing app will provide the quickest route to the user which can be alternated upon the users wish if there's any issues with the current route.
One final feature presented to the user will be the ability to use a booking a regulation system to categorise services in order to prevent conflicts and also to catalogue the amount of journeys which can also be used in statistics representation.

Firms can also use the modelling and statistics function as analytics data in projects such as recording flight data through different geographical areas in order to change or alter routes in the future if need be.
Furthermore the proposed solution would be suited towards commercial and industrial use because it could also be extended to offer marketing data and services by analysing traffic

within the area after rendering, commercial use of this could include use by scientist in order to analyse the impact of aircraft and their emissions in small geographic areas, future infrastructure planning in order avoid frequent flying paths and such. Furthermore industrial use of this application could be to place public amenities across the more used routes in order to improve infrastructure and accessibility.

## Interview Research

In the interview section precise and specific questions were asked to potential stakeholders in the Active Route Finder to figure out their interests in the program and also how they would react and use the program individually or as an organisation, from the interview process key information about how the program will be used can be translated into optimising certain features to make them more user friendly and accessible.

**Commercial Use Questions:**

These questions were designed for Giancarlo who is a pilot who performs private flights and also for leisure, he is an experienced pilot with over 2000 hours of flight time under his belt.

1. **What current system do you use for route navigation?**
2. **Do you have any issues with your current system?**
3. **Does your current system always provide the most efficient route?**
4. **What would be your ideal version of this system?**
5. **Do you have any one specific need that must be addressed or implemented?**
6. **Do you think the Active Route Finder will be able to help your efficiency?**

**Question Analysis:**

Question 1 is aimed to find out about what current technologies are in place for commercial use in situations such as piloting aircraft.
Question 2 aims to root out any existing issues with previous technologies in order to optimise the Active Route Finder to make sure that users have a better user experience and accessibility.

Question 3 tries to figure out whether the stakeholders already have access to the most efficient route using their current technologies in order to predict the possible impact this could have on future and potential users.
Question 4 focuses on user experience and what the interviewee would personally like to see improved or included, this is in the aim to optimise the experience towards the user as much as possible.
Question 5 is aimed towards creating a software that understands and responds to user needs and creates a robust and safe system that the user can depend on when in travel.
Question 6 is a very open question in order to judge whether the stakeholders are confident that the Active Route Finder will produce a significant impact on their journeys.

**Responses from Giancarlo, a private pilot:**

**What current system do you use for route navigation?**
"At the moment me and many other pilots use the built in navigation software that is supplied as a base with almost all aviation aircrafts, there's differentiations between them but they all have the same purpose."

**Do you have any issues with your current system?**
**"**Speaking for myself, the base aircraft navigation systems are limited in the fact that they only provide navigation and waypoint data and not other features, to access other features such as weather and other miscellaneous information I have to navigate to a whole other system in the aircraft which can become tiresome if I am checking the features regularly.**"**

**Does your current system always provide the most efficient route?**
**"**Flying domestically the routes can be efficient enough most of the time, but sometimes due to emergencies and the general flow of aircraft the routes can become elongated.
Furthermore the problem becomes worse when travelling globally, the waypoints can vary from being ahead of each other to varying in altitude and direction enough to add delays to the overall travel time.**"**

**What would be your ideal version of this system?**
**"**A system that obviously provides the basic features that are the industry standard but also compiles extra features into the navigation software in order to ease accessibility and make my life easier.
I believe having all the information I need compiled into one place could reduce the amount I make, which could eventually be the difference between a smooth flight or an emergency.**"**

**Do you have any one specific need that must be addressed or implemented?**
"The need for safety and robustness is key, the software must have fail safes and redundancies in order to allow safe operation of the aircraft, if one system fails another one must be accessed in order to keep safety paramount."

**Do you think the Active Route Finder will be able to help your efficiency?**
"I believe that if the necessary information is compiled into one area then decision making can be made quicker when in the sky which could help to alter the route ahead of time and increase the ease of navigation and efficiency."

# Research Solution

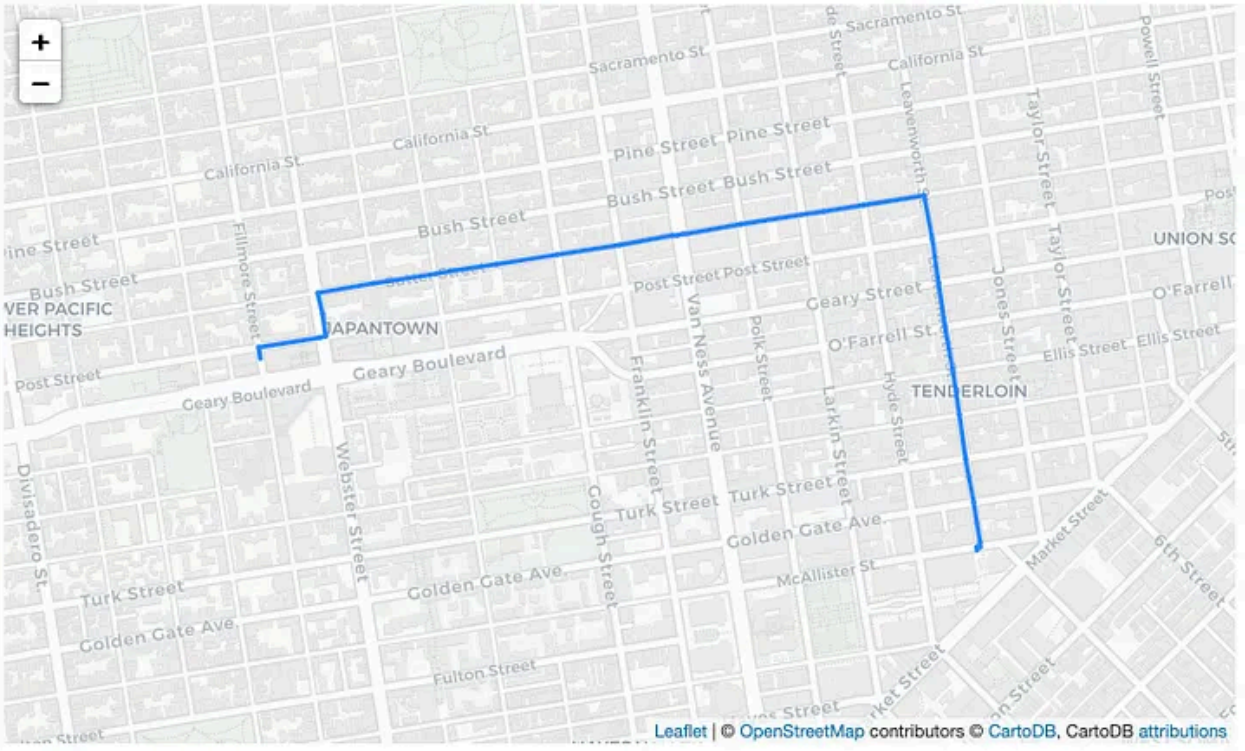
**Existing Similar Solutions:**

Other applications such as this route navigation software will use an imported data set of road network in California, a set of specific modules are imported which allow the program to read the map and return a graph 'where each node is a geographical position, and each edge contains information about the road
linking the two nodes'.


**What I can use in my program:**

The use of ‘geographical nodes’ can be implemented across the map that is inputted by the data set and mapped to road networks such as dual carriageways, motorways, slip roads and restricted access roads.
Furthermore I can use this geographical node network to create a chain of nodes that will link across road ways in order to act as directions, they will direct the user towards their destination and can also act as reference points for analytics for uses such as total distance travelled and such.

Another route visualizer uses libraries such as OSMnx which allows the use of real world map data from an open data set in order to help the basis of the software, after this the software imported another library (NetworkX) which manipulates the ‘structure, dynamics and functions of complex networks’. After this the geolocations are put into an array or database for reference.




The final result is a visual representation of the journey from point A to point B, on top of this there's functionality to change the map textures and the modes of transportation that can be used in the journey itself. Further manipulation is used by turning an address into accurate coordinates in order to connect to OpenStreetMap data and accurately plot nodes for the journey.

On top of this a static graph or map of all of the plotted nodes can be viewed for referencing and showing alternative routes that the person could take on their journey.

**What i can use in my program:**

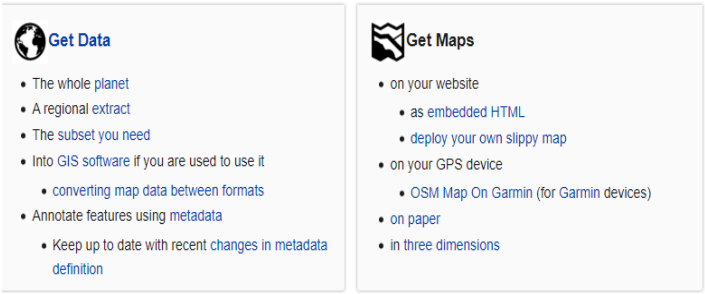
The use of OSMx and NetworkX libraries can be used in order to produce and create nodes across a geographical map which then the machine learning algorithm can access and train based off and produce rendered journey routes.
Furthermore the use of open source data from OpenStreetMap can also be potentially used as the base geographical map for which the Active Route Finder will be based off.

**Dijkstra's Algorithm:**

Dijkstra's Algorithm is the algorithm that calculates the shortest path between two nodes or points on a graph.
This can be used into the Active Route Finder in order to produce and output the most efficient route to the user, this will help to reduce the travel time as well as increasing productivity for stakeholders such as pilots.
The algorithm will be used in context with the road network as the roads will be mapped with nodes, then the algorithm will choose the nodes that will create a journey path from A to B, this will then be displayed in the form of a direction line disregarding any other nodes that aren't in the route, unless the user decides to opt for an alternative route.

**OpenStreetMap:**

Open Street Map provides an open sourced map that can be exported by an API into software projects in order to provide map overlays for different purposes.

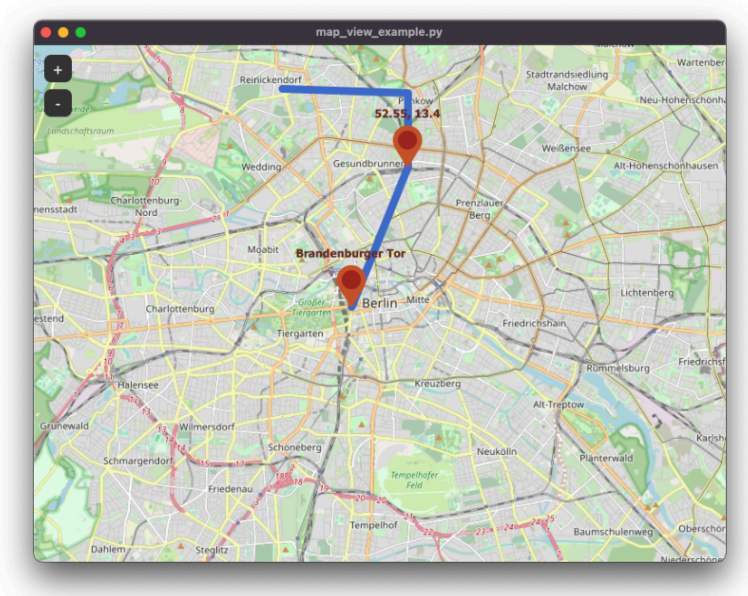


The website provides information on how to extract data and maps from the website in order to use in projects, it allows users to embed maps into code and extract real life data from maps onto different programs.

**What I can use in my program:**

The ‘Get Data’ function on the website could be concatenated with the ‘Get Maps’ function in order to create realistic waypoint information in the graphical overlay that will be provided in the graphical user interface.
Furthermore a grid of nodes could be created across the map by linking them to real geographical data which will lead to a collection of nodes that can link to Dijkstra’s algorithm in order to provide the shortest path between the departure and arrival locations either globally or domestically.
Furthermore assuming that waypoint information will have realistic information linked to them, then data such as altitude, latitude and longitude can be linked to the displayed waypoint in order to create an efficient route that maximises the fuel efficiency of the aircraft as well as minimising the time spent in the air.

**TkinterMapView:**




Here is a mockup of the tkintermapview module that allows openstreetmap to be ported to tkinter and customtktinter and displayed within the window, the map can be interacted with and markers that take in coordinates will be displayed onto the map.
This will make up a bulk of the widgets on the window in the program.

# Essential Features Of Proposed Solution

**Initial Prototype of Proposed Solution:**

The Software will prompt the user with a Graphic User Interface, where the user will pinpoint the departure address and the destination address, then the program will calculate the best route based off user inputs and suggestions, then the route will be displayed as an overlay of a map to the user so they can follow the route.
The software will also provide analytics data which an individual or even a company can access in order to measure things such as distance travelled, fuel consumption and fuel costs and also estimated travel time, this data will be presented in the form of statistics and graphs.

**Limitations of Proposed Solution:**

Furthermore speed may not be able to be detected by the aircraft when in transit so at different stages of the journey, based on the aircraft certain set airspeed values will be used in calculations for statistics and data analysis.
One final limitation of the proposed solution will be the fact that the map may have to be regularly updated in the future in order to maintain accuracy which may lead to the machine learning algorithm having to be reset so it can be trained upon more accurate data.
Djikstra's algorithm and machine learning may be difficult to achieve as the tkintermapview module may have conflicts with nodes and information collection to create a training model for the machine learning.

# Requirements for Active Route Finder Software

**Hardware Requirements:**

**The hardware requirements will outline the necessary physical components the user must have in order to run the software.**

| Requirement | Specification |
|---|---|
| Computer with two or more cores | A computer with multiple cores will allow for the program to use parallel processing in the rendering; this will allow the program to reduce the loading time for an output. Allowing a multi-core computer to run this software will drastically increase the efficiency of the software and therefore decrease the chance of errors occurring. |
| Common input/output devices | Average computer peripherals such as a monitor, mouse and keyboard will be used by the user in order to navigate the software. As well as allowing the user to interact with the software through functions such as start/end journey and calculating fuel consumption costs. |
| Minimum 4GB RAM | The software will need a minimum of this amount of RAM in order to provide the concurrent processing as well as the processes that will be performed such as data storage, outputs and graphical updates. |
| Minimum processor speed 2GHz | A minimum processor speed for 2GHz will ensure the software can handle instructions and calculations on demand without any lag or buffering that may ruin the user experience. |

**Software Requirements:**

**The software requirement will outline the necessary features the user must have installed.**

| Requirement | Specification |
|---|---|
| Windows operating system | These will be used as OS in order to run the processes and access to the program.<br>As well as compiling the code the device will have to have python downloaded onto it so the processes can be carried out by the computer and then displayed to the user. |
| Python IDE or compiler | The project's native language will be in Python so a compiler will be needed in order to run the code.<br>Furthermore the compiler will also be needed in order to display information from the software onto the users device |
| Python libraries | Specific libraries and modules will be imported to assist with the main functionality of the route finder, libraries for example ‘date/time’ and ‘random’ libraries can assist in the features such as travel time estimation and fuel consumption calculations. |
| Common software/hardware drivers | These drivers will be needed to allow the peripheral devices to communicate with the software on the computer to allow the user to interact with the active route finder in multiple different tasks. |

**Solution Requirements:**

**The solution requirements will include the main aspects of the software that need to be included into the solution.**

| Requirements | Specification |
|---|---|
| Interactive graphical user interface | This will allow the user to interact with the program without the need of manually typing into the command line interface, therefore improving the user experience. As well as providing information to the user. |
| Integrated analytics system | The built-in analytics system will analyse the journey from the departure to the destination and then if the user requests, provide them with statistics and graphical data on factors such as distance travelled, fuel consumption and cost as well as the estimated time of travel. |
| Graphical map overlay | This physical representation of the journey will be represented to the user as an overlay on top of the current waypoint networks map, the user will refer to this during their journey. |
| Continuous user support | Complete user manuals in order to guide the user towards learning how to use the software and how to use features integrated within it such as the analytics system. This is aimed to improve user accessibility and provide user support |
| Journey render produced within under 1 minute | This will keep the efficiency standards of the program high in order for customer satisfaction and to keep the software relevant for future use such as in emergent times. |
| Software double checks address is valid against user request | This will ensure data validity and that the wrong address or an invalid address is not inputted into the software which could potentially impact the journey render. |

| | |
|---|---|
| Alternate route render upon user request | Incase the user encounters any congestion or obstacles in the initial route, they can opt for an initial route to be drafted and if necessary individual nodes can be selected to be rerouted around |

## Success Criteria

**The success criteria will outline what criteria the software is to achieve in the form of measurable goals that can be achieved.**

| Success Criteria (Functionality) | How it will be achieved |
|---|---|
| Functions to start and end journey | Clear graphical user interface interactive buttons with labelling in an area where it will be easy to access by the user |
| Accurate overlay representation of journey | A mini view of the global network map will be shown to the user with connected nodes forming a journey path which the user references to, if selected the nodes will be able to represent waypoint information such as their number. |
| Accurate node waypoint information | Each node will be given waypoint information such as numbers for which the user can refer to, the node numbers will be in order starting from the departure address |
| Accurate and up to date global air network map | The most recent and updated global network map will be imported in order to provide accurate information to the user |

| | |
|---|---|
| Concise statistics and data representation | A separate tab will contain functions for which the user can access in order to calculate key information such as distance travelled, fuel consumption and cost etc. |
| Suitable machine learning algorithm with a data set to be trained upon | A machine learning algorithm will be used that can be inputted with numerical values or graphical data in order to allow the model to train itself upon data given to it. |
| Appropriate data handling | Ensuring that the software handles data in a simple and effective manner that is easily understood by the user and reduces conflict within the programs processes |
| Accurate and functional user interface components | The map overlay and widgets on the user interface play a big part of the user experience and accessibility and therefore making sure they work correctly ensures the user can have the best experience on the software without experiencing any issues. |

| **Success Criteria (Usability)** | **How it will be achieved** |
|---|---|
| Hovering or clicking on a node reveals it waypoint information | This will be achieved by connecting a function to the label button in order to display the necessary information |
| Easy to navigate the user interface | This will be achieved by clearly displayed buttons as well as accurate user manual information in order to help the user navigate the software |
| Easy to understand user interface | This will be achieved by using a minimalist |

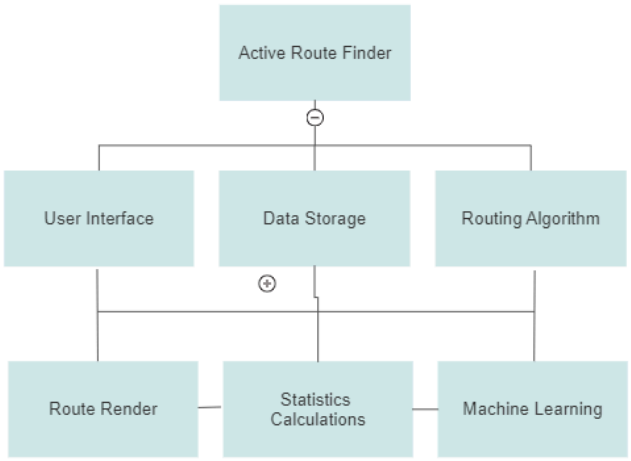
| | |
|---|---|
| | user interface, this will allow for a less cluttered user interface which will allow for easier navigation around the software |
| User manual | By providing a tab to clearly and accurately explain to the user common functions and how to use the software such as in terms of inputting addresses |
| Efficiency of use | The software must be efficient for the user and allow them to productively plan routes and gain useful information from the routes so that they can |
| Lag free user interface | Ensuring the smoothness of the user interface will help to control the user experience allowing for greater accessibility and user onboarding. |
| Customisations for accessibility | The software will include user interface scaling and theme modes for the software so that users can change the software in order to meet their own needs. |

# Design

## Problem Decomposition

The problem decomposition of the initial problem will allow for specific problems to be matched with suitable computational solutions as well as breaking down the main problem down into smaller tasks and problems to make it more manageable to create, and therefore be presented in a way that creates a clear layout for the direction of the software.

**This is a hierarchical diagram for how the Active Route Finder will be decomposed:**




**The main problem can be broken down into its main aspects and functionalities:**

- **User Interface**
- **Routing Algorithm**
- **Statistics Calculations**
- **Rendering and Displaying of Final Result**
- **Machine Learning through Multiple Renders**
- **Storing of Data**

These elements of the main problem create the foundation for which decomposition will be laid upon by carefully analysing each element and creating a tailored solution for each one that will perform the most efficiently as well as cooperate with the other elements to create a final solution/product that will satisfy the stakeholders needs.

**User Interface:**

The user interface will be divided into three sections which each on their own and together will contribute to the user experience, the main functionality of the user interface will be to provide an environment for selecting specific functions such as the statistics category which will include fuel consumption costs, distance travelled and estimated travel time. Furthermore the user interface will include a booking and regulation system that the pilots can use to create a schedule for their day in order to prevent a conflict of services.

The second section of user interface will be the connectivity between the designs that are shown to the user on the screen and the function behind the interactions they will carry out. This will take example in events such as the statistics calculation and route calculation functions, the user will be able to interact with a physical interface referring to the functions and behind the user interface algorithms and calculations will take place in order to create an output to the user which will be the third section of the user interface.

The third section of the user interface is solely based on data representation and displaying results to the user. This will take action in the data representation of the statistics module and the display of the route rendered to the user.
The use of the data representation is to provide the output and data from the software in a form that the user can easily understand and interpret in order to solve the initial problem of travelling from one place to another.

**Routing Algorithm:**

The routing algorithm's main purpose will be to extract data from the nodes corresponding to the map and use user inputted data to plot the most efficient route between locations that the user has inputted into the software.
The routing algorithm will access from a collection of geographical nodes that have been plotted across maps and specifically on waypoints for aircraft, the nodes will then be connected through the algorithm to create a route of nodes, features will be added to the routing algorithm to exclude certain nodes and create an alternative route for the user should anything be wrong with the initial route.
This could possibly be integrated with the machine learning algorithm to then create a more efficient and accurate render every time the software is subsequently used.

**Statistics Calculations:**

Combined from data storage and the routing algorithm elements, statistics calculations will pull data from them to create data sets with key information that the user can use to improve their efficiency as well as keep track of their services.
Examples of these calculations will be to calculate fuel consumption, this will be concluded by asking the user to input information about their aircraft, if not the average fuel consumption for that aircraft class will be used, it will then be compiled with the amount of distance travelled to then create an estimate for the fuel consumption of the users aircraft.

Distance travelled could be calculated by introducing a scale to the map and then calculating the estimated distance between nodes to then compile them into a total distance travelled value, this will be used in conjunction with fuel consumption calculations.
Estimated travel time can be calculated with the estimated distance travelled and an average speed that will be used for the whole journey, the travel time can then provide guidance to the user about the composition of the journey and expectations of it.
One final statistical calculation that will be presented to the user will be fuel costs, this will pull data for the current or average fuel price and then combine with the fuel consumption to calculate the cost of the fuel that was used on the journey.

**Rendering and Displaying of Final Result:**

A map overlay will be presented to the user which will contain a graphical representation of the map itself and a highlighted route across the map from the arrival address to the destination, furthermore the map overlay will be provided by an open API which will connect to the software in order to provide functionality to the user and for interactivity.
Once the final result is displayed functionality such as interacting with the map will still be enabled, therefore allowing the user to have greater control and accessibility to their journey and preferences.

**Machine Learning through Multiple Renders:**

One feature of the software is to be ever-evolving, this will be implemented by using a machine learning algorithm to analyse routes that the user has inputted previously and the result from them to then create a more accurate and efficient route everytime the software is used subsequently.
In order to supplement this a form of data storage will have to be linked to the machine learning algorithm so that it can access this file with previous route data and then use it to further train its algorithm.

**Storing of Data:**

The final element of the software relates to statistics calculations, features such as the reservation and booking system and the possible machine learning algorithm ; data from these elements will be stored in different forms of structures, which can vary from arrays to tables, dictionaries and databases.

The plan is to store this data externally on the users device in the form of word files and etcetera, data from these files will be read and written from and to the software depending on the function that is being processed.
The data storage system will have been designed to be accessed quickly and efficiently by the software in order to increase the efficiency of the software and also the consumer experience.

Dividing the software into sections will improve the decomposition of the program and provide sectioned aspects of the software to elements of the program depending on their function and how they supplement the software and each other.
Furthermore decomposing the software into smaller section and therefore problems, the solutions can be easily identified and therefore created section by section without having to worry about a large problem instead, this will eventually improve accessibility to maintenance as the solution has been sectioned into different elements allowing easier maintainability as only one area is to be worried about and not the whole code.
Eventually the six elements of the Active Route Finder come together in order to create a finished software that communicates between its different components and elements in order to produce the best user experience possible.

**A Initial Rough Design of the Elements on the User Interface:**

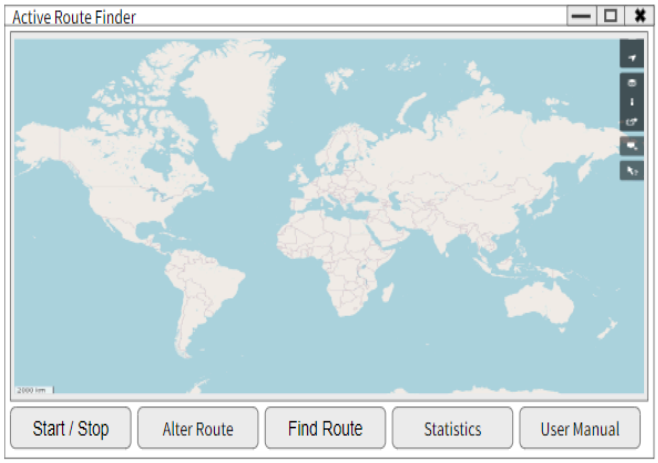
**Success Criteria for the User Interface:**

- **Clear and easy to understand layout,** this will allow for a smooth and simple user experience that can improve customer satisfaction with the finished software and help with accessibility over a wide range of users
- **Map render displayed in the main window,** this is important in order to provide easy access to the main functions and displays of the software. It must be displayed somewhere it can be accessed and viewed easily. Furthermore having the map render being displayed in the main window will allow for easy navigation as the user will be greeted with the map information instead of having to navigate through different pages which may confuse the user and lead to user dissatisfaction.
- **Clear display of secondary options,** this will include the functions such as the user manual, statistics calculations, start/stop of the route algorithm, the alternate route function and most importantly the find route algorithm function that the user will proceed to in order to input the route information. As a result displaying all of these secondary functions on the side of the main window will ease navigation for the user and improve user accessibility and satisfaction as almost all of the functions are within one button's reach.

# Structure of Solution

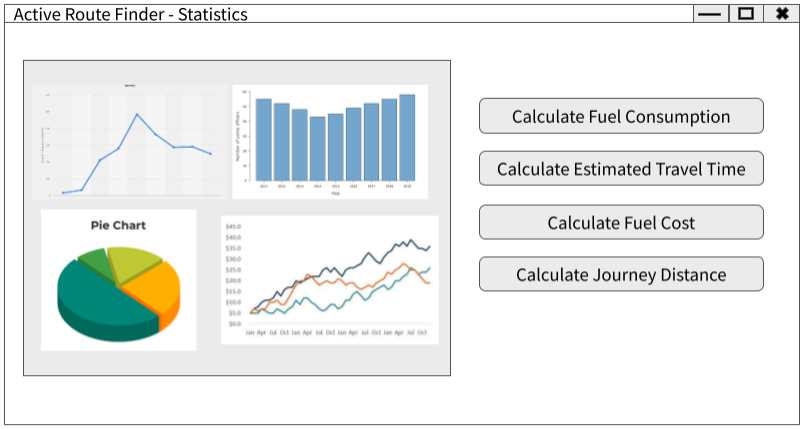
The main user interface will provide access to the most important functionality of the software as well as displaying the map information and routing information.
In terms of user design in mind the main focus for each window

**Updated User Interface Design of Software**

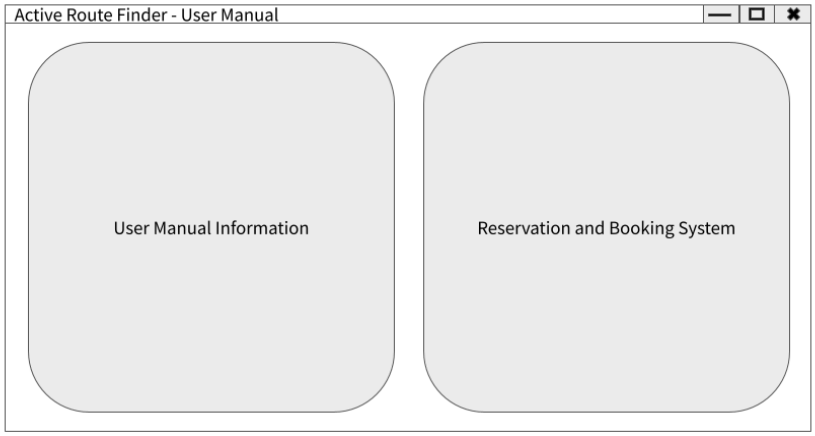



**Statistics Module**

In this window that can be accessed through the ‘statistics’ function the array of calculation functions will be stored and also displayed.




**User Manual Module**




Once compiled the structure to the solution in terms of user interface will provide a clean and accessible experience to the user in order to improve stakeholder satisfaction and ensure the experience of using the Active Route Finder is as optimised as possible.
In order to achieve this, multiple user interfaces ( the prototype designs of which have been added above ) will be created through the ‘tkinter’ library in python.

‘Tkinter’ will provide access to graphical user interface commands and designs that the software will incorporate in order to provide a viable user interface and experience.

Overall the structure of the solution will combine multiple user interface elements with programming solutions from the decomposed elements to create a functional user interface that will respond to user interactions and output renders and statistics information to the same user interface for easier usability to the user and increased accessibility.
In order to link the user interface functionality such as button and input boxes to software algorithm and components, the button and input box functions will contain global variables and other subroutines such as the routing algorithm, route render display code and etcetera. This connection between two aspects of the software will coordinate with each other to provide a functional software that the user does not have to use a command line interface in order to operate which can be tedious and lengthy.

**Algorithmic Structure of Solution**

The Active Route Finder software will compile different types of algorithms such as possible machine learning, data analysis, calculation and data storage algorithms in order to produce a finished product that will satisfy the pilot and their needs.

**Structure of Machine Learning Algorithm:**

Simply, the machine learning algorithm will be self-sufficient and continuously learning from data from previous journeys, the algorithm will provide a strong backbone to the software by providing a data processing event that will eventually become more and more accurate to increase the overall validity and accuracy of the route renders.

# Algorithm Design

**Overall Algorithm Design:**

Take inputs from user on location information > Output a rendered navigation route to the destination address.
The main, primary solutions that must be implemented for the Active Route Finder will be the machine learning and route navigation and the data storage algorithm. These three algorithms will make up a bulk of the softwares usage as it serves the primary purpose of providing routing information for the user.

**Algorithm Research and Development - Design:**

In this section of research the basis for the final algorithm will be conducted across different programs that have had a similar goal and basis of problem solving will be used in order to influence the algorithm design and the implementation of it into the software.

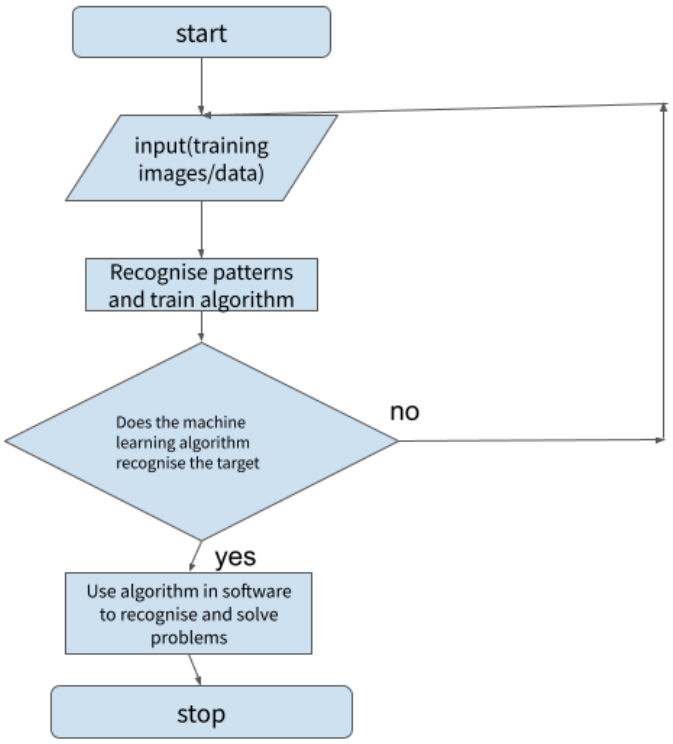
The propositions for the main algorithms that will inputted into the software can be divided into 3 sections:

- **A possible Machine Learning Algorithm**
- **User based Dijkstra's Algorithm**
- **Node Algorithm**

**Research for the Machine Learning Algorithm:**

The main focus of a machine learning algorithm is to train itself based on given data inputted into the algorithm in order to solve problems from this inputted data, the machine learning algorithm will constantly 'train' under this data in order to become more and more accurate everytime it is tasked with solving a problem. One example of this could be a machine learning algorithm being inputted with images of different shapes in order to select and identify a different array of shapes from a separate inputted image.

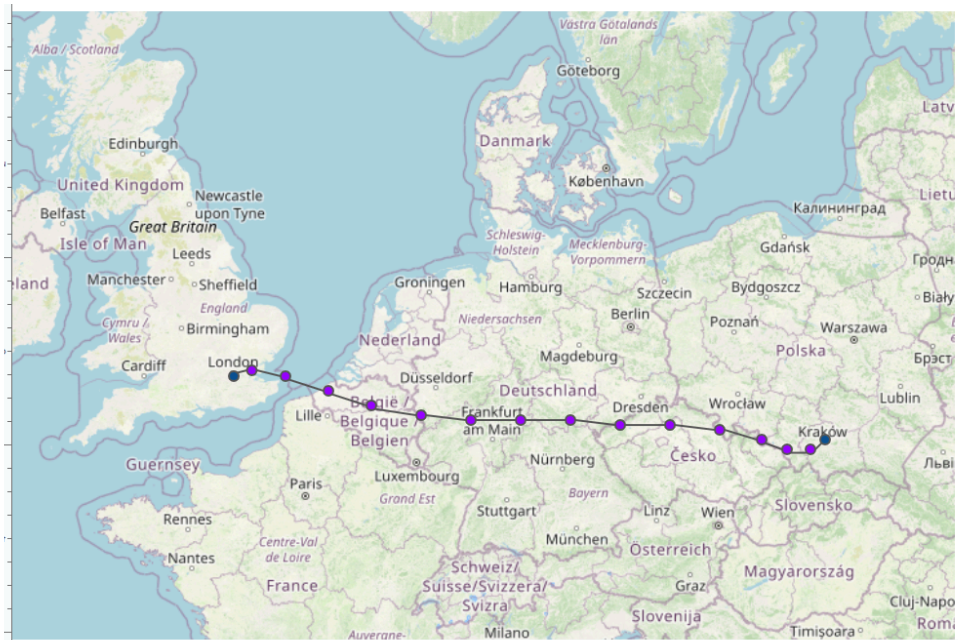
The flowchart below demonstrates how the machine learning algorithm will be implemented and designed, the training data in context of the active route finder could consist of global maps and aviation maps in order to differentiate between waypoints and other landmarks on the map, through this the machine learning algorithm will plot an accurate and suitable route towards the destination address.




Furthermore the algorithm will always be self-learning depending on the condition, it will use previous events and route calculations to continuously update and become more accurate when rendering the route overlay onto the map for user access.

The type of machine learning algorithm to be used will be a support vector machine algorithm and a decision tree algorithm.
The support vector machine algorithm will be used to analyse the nodes of geographical locations that are inputted into the machine learning algorithm and then analysed

**Dijkstra's Algorithm:**
Dijkstra's Algorithm in the Active Route Finder will be used in the context of connecting nodes from the departure location to the destination location via the most efficient route using the least amount of nodes to connect the two locations.
Nodes between the departure and destination location will be referred to as waypoints, they will be linked to geographical data such as altitude, longitude and latitude.
A chain of connected nodes will then be outputted to the software which will then be provided to the user in a readable and understandable format on the graphical map overlay as well as in words to improve user experience and understanding of the software.




This is a mockup of what the Dijkstra's algorithm ( London to Krakow ) in work would like: the path is not likely to be a straight line due to various node placements as not all waypoints can be placed anywhere such as heavily residential areas or restricted air spaces.

**Pseudocode for Dijkstra's algorithm:**

```
nodemap.add_nodes_from(list/array of precompiled attributes)
nodemap.add_edges_from(list/array of precompiled attributes)
```

First of all, to implement Dijkstra's algorithm into the route navigation section of the active route finder, the node algorithm must be compiled with it in order to create a completed section of code that references each other.

**Node Algorithm Design:**

```
# library imports
import network noding library
import graphing and visualisation library

# in the context of the active route finder, nodes will be referred to as waypoints

# create empty base for waypoints
Graph = graph function

# add basic waypoints
Graph add node (1)
Graph add nodes from (lists/array)

# adding connections between waypoints
Graph add edge (1,2)
Graph add edges from (list/array)

# creating waypoint data that can be admitted to software for data storage and analysis
total waypoint list = Graph nodes function
total links list = Graph edges function
total links to specific waypoint = Graph adjacency function
total links between waypoint = Graph degree function

# adding graph attributes
Graph = graph function (graph description = "active route finder map")

# waypoints will be attributed to unique 4 letter codes

# adding waypoint attributes
Graph nodes [1]["unique identifier"]
Graph nodes[1]["altitude"] = "00000000"
Graph nodes[1]["longitude"] = "00000000"
Graph nodes[1]]["latitude"] = "00000000"

waypoint data = Graph nodes data function

# creating a weighted graph
Graph add edge(1,2)
Graph[1][2]["weight"] = 100

weighting/edge data = Graph edges data function

# displaying the graph data

network x draw function (Graph, with labels = "true",
display edge weight = network x get edge data function(Graph,'weighting')
network x draw edge labels function (Graph, position, edge labels))=
visualisation show graph function

# this will allow the visualisation library to output the graph of waypoints with accurate waypoint attributes
```

This pseudocode defines the base structure of the python code that will create the basis of node function, the purpose of the code will be to create a set of nodes in parameters around the departure and destination locations on a map.
After this Dijkstra's algorithm will analyse the nodes individually and as a group in order to determine the most efficient route from departure to destination.
The pseudocode sets out the libraries that will be used, one of which will be the NetworkX library that provides node functionality and visualisation as well as matplot library, a mathematical visualisation library that produces graph plots and etcetera.
Furthermore the pseudocode demonstrates how waypoints can be paired with attributes and information such as altitude, longitude, latitude and importantly a unique identifier for each waypoint along the route.

The route can be outputted with a weighting as well which during testing and development will come into hand and may also be used for further statistical analysis and machine learning on the route waypoints and their relations to each other.
In terms of waypoints, they'll be added in mass across the area of travel then in combination Dijkstra's algorithm will single out the optimal waypoints to be connected to form a route, the waypoint will then be connected and visualised on a graph on top of the map and outputted to the user

```
# algorithm for creating geographical nodes

# import network library
import networkx as nx
import matplotlib as plt

# in the context of the active route finder nodes will be referred to as waypoints

# create empty base for waypoints
nodemap = nx.Graph()

# add basic waypoints
nodemap.add_nodes_from(list/array of precompiled attributes)

# adding connections between waypoints
nodemap.add_edges_from(list/array of precompiled attributes)

# creating waypoint data that can be admitted to software for data storage and analysis
allwp = nodemap.nodes
alledge = nodemap.edges
adjacency = nodemap.adj
links = nodemap.degree

# adding graph attributes
nodemap = nx.Graph(graph_description = "route map")

# waypoints will be attributed to unique 4 letter codes

# adding waypoint attributes
nodemap.add_nodes_from([(1, {"unique identifier"})])
nodemap.nodes[1]["altitude"] = "00000000"
nodemap.nodes[1]["longitude"] = "00000000"
nodemap.nodes[1]]["latitude"] = "00000000"

wpdata = nodemap.nodes.data()

# creating weighted graph
nodemap.add_edge(1,2)
nodemap[1][2]["weighting"] = 100
wpw = nodemap.edges.data

# displaying graph data
nx.draw(nodemap, with_labels='True')
edge_weight = nx.get_edge_attributes(nodemap, weighting')
nx.draw_networkx_edge_labels(nodemap, pos, edge_labels = edge_weight)
plt.show
```

This is a more refined design from the pseudocode and converted for use in python.
This code, although not fully compiled yet, will be referenced for later use in the programming process.

Incorporating this code into my software will be achieved by the code being used across the route navigation sections of the Active Route Finder and this will create a function that can be called throughout the software where needed in statistical analysis or even the route navigation and machine learning aspects.
Incorporation aspects will include global variables that can be ported across the whole software in order to be used in other subroutines such as route calculations and machine learning aspects.
Local variables will also be implemented into the node algorithm that are differentiated from global variables in order to keep consistency as well as readability when it comes towards coding the software itself.

**Pseudocode for entire software:**

```
Import mapping libraries and interface modules
Class app ():
    Procedure (user interface):
        Interface = module size and position of window
        Formatting Interface = positioning between widgets
        Widgets = buttons, entry fields and cursor events
    End procedure
    Procedure (map overlay):
        Formatting = size and position of overlay on interface
        Map info = map tiles, starting position of map and default map zoom
    End procedure
    Procedure (Interactions):
        Cursor event = place and recording nodes on the map overlay
        Button event = clearing nodes, resetting route
    End procedure
End class
User interrupts for exiting route finder causes route nodes to be reset
```

This iteration of the pseudocode for the software shows how the software can be split into 3 main subroutines within a parent class, one procedure will take care of the user interface, one procedure will take care of the map overlay display and the final procedure will take care of user events.
Integrating the subroutines within a parent class will allow for modularity to be integrated, furthermore once the parent class ends exiting the active route finder will remove the

information for previous routes carried out so that the information from the previous routes does not keep on backing out to the display for the user.

**Pseudocode for the events:**

This block of pseudocode identifies how software events will be reacted to by the program as well as defining the data structures that will be used in conjunction with the map overlay and info output system.

```
Pseudocode for events:
//initialise lists for data handling
//lists will be global to work across whole program
List1 = list for nodes coordinates
List2 = list for unique identifiers
//list 3 will be a tuple
List3 = concatenate list1 and list2 into one item as a tuple
//unique identifier creation
Identifier = random 4 letter uppercase string
Append identifier to list2
//list manipulation from events
IF map overlay event occurs THEN
        Retrieve node coordinates from selected location
        Append node coordinates to list1
        Take index of node and get corresponding identifier index item
        Retrieve both node coordinates and corresponding identifier and add to list3 as tuple
End IF
IF node delete OR route delete occurs THEN
        Either remove most recent node from lists or remove all nodes and all list items
End IF
```

Firstly the lists are initialised that hold the data from the map overlay and such, more lists will be added later on in development for other features.
The lists are also concatenated to create a tuple, the purpose of this tuple is to merge the lists so that the output of the node information is more streamlined and easier for the user to interpret.
Furthermore one thing to mention is that the data structures have a global scope allowing them to be accessed from other separate subroutines for any auxiliary functions so that the list does not have to be passed as a parameter through all of the subroutines.
Furthermore functionality to remove nodes from the corresponding lists has been included so that the user accessibility functions are included within the pseudocode.

**Pseudocode for individual subroutines:**

**ClearLastNode Subroutine:**

```
class arf
        private procedure clearLastNode(arf)
                remove nodelist last index
                remove idList last index
                remove totalList last index
                delete most recent waypoint
                delete most recent waypoint path
        endprocedure
endclass
```

This pseudocode defines the procedure for the clearLastNode subroutine, this subroutine is placed within the arf class and uses the class as its parameters as the parent class contains the overlay map and its functionality.
The functions within the subroutine are activated once the subroutine will remove the most recent node information from all the appropriate lists, tuples and overlays.

**PathConnect Subroutine:**

```
class arf
        private procedure pathConnect(arf)
                append pathList with currentNode information
                output updated pathList to command line
        endprocedure
endclass
```

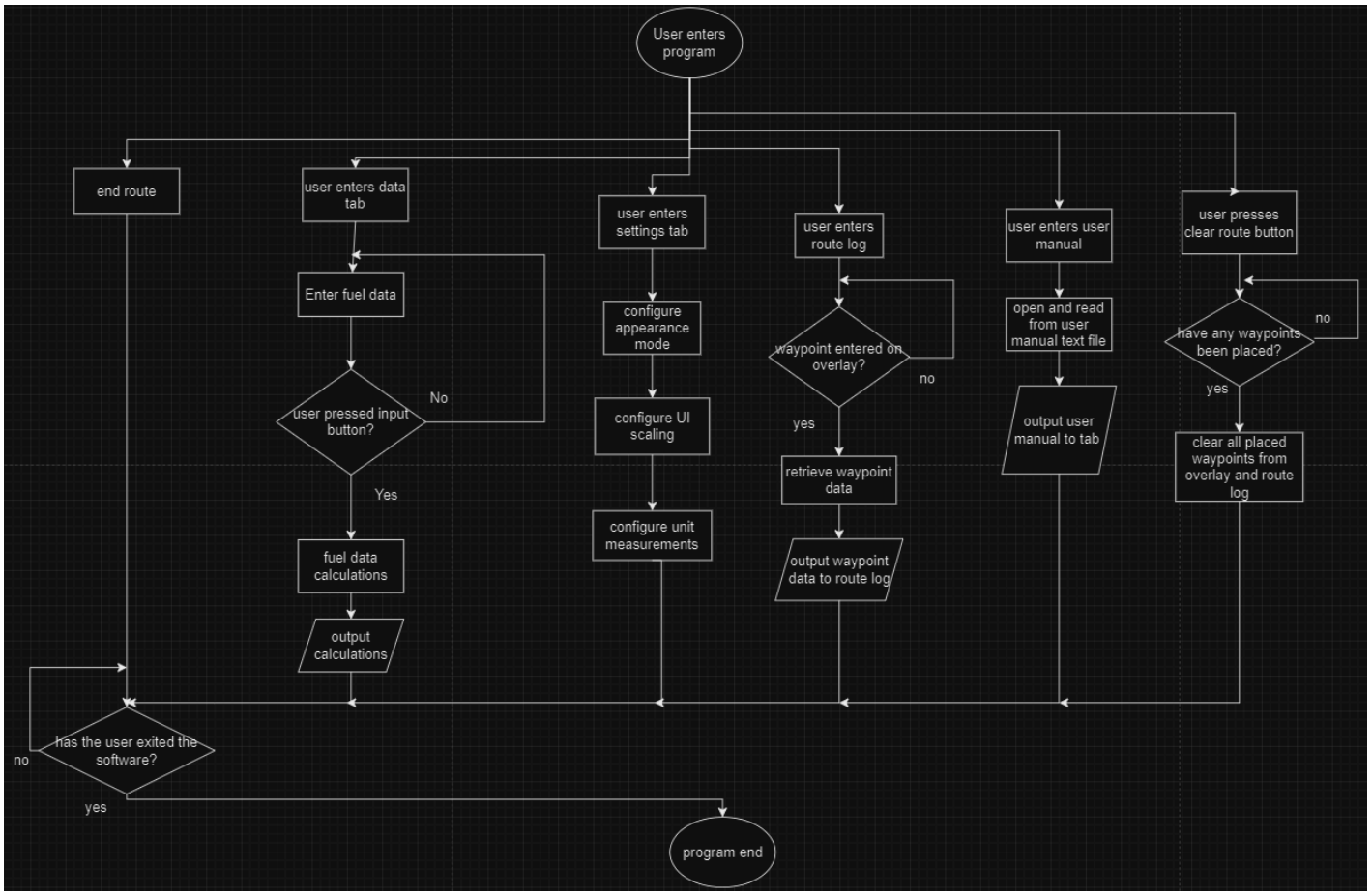
The pseudocode here represents the pathConnect subroutine that is activated from within the active route finder class every time an update is made to the map overlay on the user interface, the subroutine includes functionality to connect the 2 most recent waypoints together with the tkintermapview path function allowing a continuous path to be made between each and every single node placed onto the map overlay.
The pathList is also updated along with the new waypoints and printed to the command line for testing purposes.

**Validations Subroutine:**

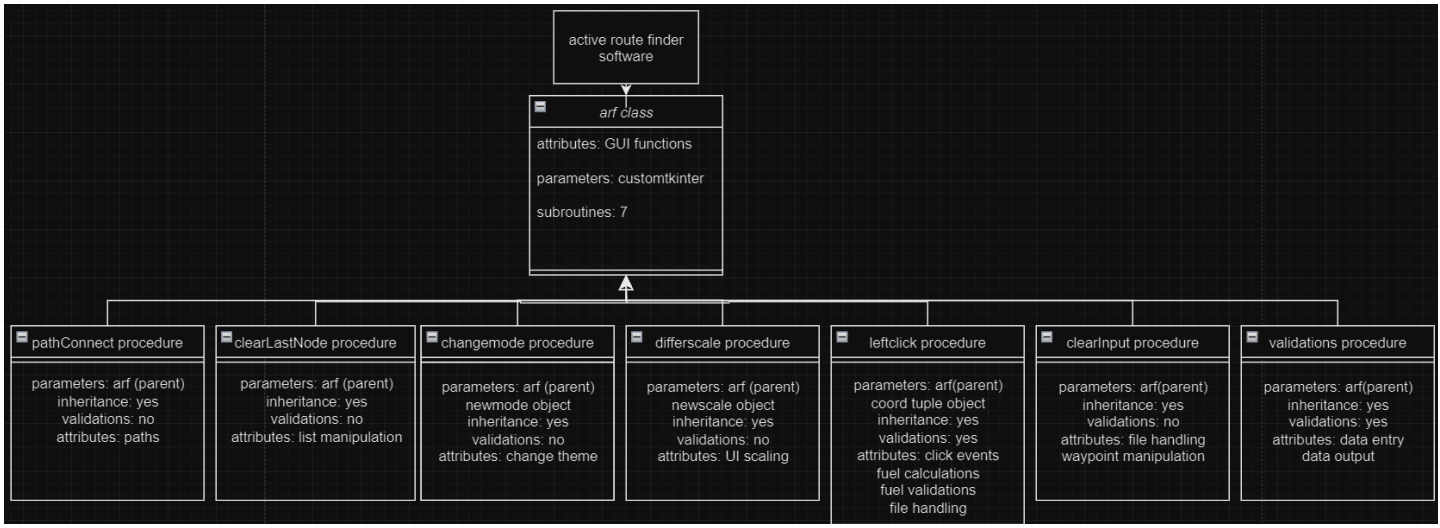
```
class arf
        private procedure validations(arf)
                try:
                        // data sanitisation and validations
                        inputData = pull data entered to entry fields
                        output data to command line for testing
                        if string length < 6 then
                                output input too short
                        if string length > 6 then
                                output input too long
                except:
                        output invalid fuel price input
                try:
                        conInput = pull data from entry fields
                        output data to command line
                        if conInput > 4 then
                                output input too large
                        if conInput > 4 then
                                output input too small
                except:
                        output invalid fuel data
        endprocedure
endclass
```

The validations subroutine allows the inputs into the data entry field to be validated and sanitised before being entered into calculations in order to ensure that valid outputs will be produced within the correct ranges.

**Flowcharts:**




This diagrammatic flow chart for the active route finder splits the program into different sections and annotates their flow of processes into a drawn form.
Every process begins with the user entering into the software and ends with the user closing the software, the software branches into different sections based on where the user selects, and the end route section will eventually connect all the other sections so that the program can be closed at the end of its use.
There are multiple instances of branching within the software at areas in the program such as user inputs and events and recursive functions that continuously check the state of certain components, furthermore iteration is common when being applied in circumstances such as list manipulations.




Here are the class diagrams for the active route finder, one main class will encompass 7 smaller subprocedures that will inherit from the parent class. In this case no polymorphism will be used, in the subroutine descriptions are the parameters and attribute descriptions of the subroutine as well as if there's validations and inheritance present.
Since every single subroutine is instantiated within the arf class they are all connected to the class diagram.

## Usability Features

In the active route finder software the main program will consist of around 9 usability functions that the user can use in order to achieve their objective of rendering a route from a departure address to a destination.
The functions include primary functions which will provide the main functionality of the program to the user alongside auxiliary functions that will provide support and quality of life improvements to the users accessibility and usability of the software.
The primary and auxiliary functions will work collaboratively to produce a finished product that will provide routing information to the user at a moment's notice.

**Primary Functions**

**Start / Stop Route:**

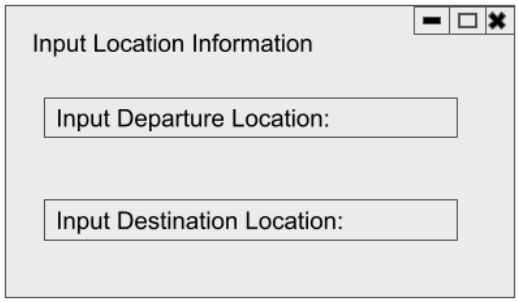
This function will be displayed on the main window and will be in charge of beginning and ending the route display to the user when the software is in use, this will be used when the user has already inputted address information into the software and wants to start the route rendering or when they have made use of the route render and wish to end the function. This could also be used to input a new route after one has just been completed.

**Alter Route:**

This will allow the user to manipulate and change  the current set route that is displayed, this could be due to physical obstructions on the original route, such as traffic, construction works or accidents, in this case the user will be able to mark a section of the route which they want to alter and the software will create another efficient route around the obstruction.
This function will improve the accessibility of the software allowing the users to continue to their destination even if there is an obstruction in the route without having to recalculate the entire route from beginning to end.

**Find Route:**

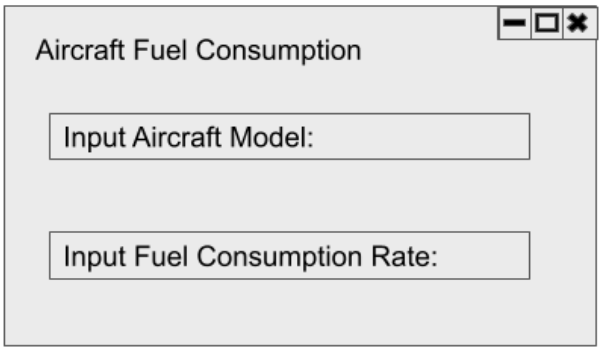
This function will provide map data for the user and then allow them to input data into the software through several input fields. Inputted data for example could be destination address, departure address.
Through these inputs the software will compile the waypoint networks in order to create a route that is the most efficient and therefore has the least distance.
This is a mockup of what the basic layout of the input interface will look like, aesthetics and smaller details will be implemented later to keep the facade of the software up to date.




**Auxiliary Functions ( Statistics Section )**

**Fuel Consumption Function:**

The fuel consumption function will implement the user aircraft data alongside calculation algorithms to output how much fuel the aircraft will consume throughout the journey, this calculator can help the user to calculate their fuel consumption which can then be correlated with the fuel cost to produce a figure for the fuel consumption and cost for the journey, these statistics will be helpful on the business side of using the software as the user can track expenses for their customers rides, the consumption will correspond to the distance of the route travelled so it will link with the distance function as well.
Below is a simple prototype design of what the user interface for inputting the fuel consumption information will look like, after this the data will be stored in a separate file which it can then be read from in the future in the form of displayed statistics.




**Fuel Cost Function:**

The fuel cost function will use an open API to pull fuel costs for the United Kingdom into the software where they will be collaborated with the fuel consumption function in order to produce a final result which will display to the user as the fuel cost for the entire journey.
As mentioned before this calculation can be used in order to calculate and track expenses for the user.

**Journey Distance:**

The distance of the journey travelled will be available for the user to view and keep track of, this may come useful when having to calculate total mileage for private business terms for customer use and transportation. Furthermore the journey distance function will be displayed in the form of statistics representations such as bar charts etc for each consecutive journey.
**Estimated Travel Time:**

This function will connect to the journey distance in order to calculate an estimate for the total journey, this will benefit the user as they can use it in conjunction with the reservation and booking system in order to plan and book services in order to have a clear and scheduled day.
Overall this feature is meant to increase the user experience by allowing clear scheduling that will help to prevent any conflicting services from occurring

**User Manual Information:**

This will be an essential feature in terms of onboarding onto the software and guiding the users and customers on how the software can be used and any extra features that are integrated that can improve the user experience of the software.
Specific features that will be included in the user manual information will include documentation on how to input information into the software such as addresses and locations, the main purpose of the software, how to use statistics modules and how they can benefit the users business, and also how users can time manage their journeys in the reservation and booking system.
This feature aims to increase the accessibility of the software by increasing the knowledge the user has of the functionality of the software and how to operate it.

**Auxiliary Functions ( Miscellaneous )**

**Reservation and Booking System:**

This feature as aforementioned in many other functions will be the sole time management and scheduling service for the user.
This will allow the user to input service information into a simple diary that spans across different dates and time, one of the main purposes of this is to improve the quality of life for the features by reducing conflicts in waypoints and therefore providing clear and reliable information to improve the overall quality of the pilots service.

## Key Variables

**Aircraft Variable:**

| |
|---|
| **Variable Name:** aircraft |

| |
|---|
| **Data Type:** This variable will be presented as a class and referenced throughout the software, however it will contain many separate integer variables that will support the ‘aircraft’ class. |
| **Description:** The function of this variable will be to contain information about the aircraft that is being used on that route and to correlate that information with other variables and data structures, as well as contain all important aircraft variables in one area in order to create a functioning software that performs accurately based off of the aircraft that is being used. |

**Fuel Consumption Variable:**

| |
|---|
| **Variable Name:** FuelCon |
| **Data Type:** This variable will be a float value of the consumption rate of the vehicle. |
| **Description:** The fuel consumption variable will play an important variable in further calculations including route calculations.<br>The variable will be derived from imported information about the aircraft and will be used in other calculations such as calculating fuel cost for the whole journey, limiting the total route distance in correlation with the maximum flying distance of the aircraft. |

**Fuel Required Variable:**

| |
|---|
| **Variable Name:** FuelReq |
| **Data Type:** This variable similar to the fuel consumption variable will be a float value. |
| **Description:** This variable will output the amount of fuel required in kilograms to the software and the user and be correlated with the aircraft information in order to create a safe route that the aircraft can fly and symmetric information to the user on how much fuel to use.<br>On top of this a small addition of reserve fuel will be added incase of emergencies. |

**Fuel Cost Variable:**

| |
|---|
| **Variable Name:** FuelCost |
| **Data Type:** Once again this variable will be a float value. |
| **Description:** Different information such as the fuel consumption, distance and requirement will be compiled to create a final fuel cost value based on current fuel prices that the user will be able to access to improve their quality of life experience. |

**Nodemap Variable:**

| |
|---|
| **Variable Name:** nodemap |
| **Data Type:** This variable will be represented as a string that holds the function ‘nx.Graph()’ |
| **Description:** nodemap will be the base variable for the node algorithm and will control different aspects of node manipulation such as adding and mapping nodes, as well as creating an empty node graph for waypoints to be plotted onto. |

**Data Structure Variable:**

| |
|---|
| **Variable Name:** nodeList |
| **Data Type:** List |
| **Description:** The node list will contain all of nodes coordinates that have been placed onto the map overlay in chronological order, the list will be linked to many important functions such as calculating the distances between nodes, the fuel costs derived from the distances as well as outputting the chronological node order for the route to the user. |

**Data Structure Variable:**

| |
|---|
| **Variable Name:** idList |
| **Data Type:** List |
| **Description:** This list will contain the unique identifiers for each node and each random unique identifier produced will be linked to a corresponding node, this list will be outputted with the nodeList to the user to show the nodes in the route as it progresses. |

**Data Structure Variable:**

| |
|---|
| **Variable Name:** distList |
| **Data Type:** List |
| **Description:** The distance list will contain the values for each independent distance between each node, the sum of the this list will then be calculated and outputted to the user in order to show them the overall distance of the route, the values will be updated |

| |
|---|
| every time a new node is placed onto the overlay. |

**Data Structure Variable:**

| |
|---|
| **Variable Name:** totalList |
| **Data Type:** Tuple |
| **Description:** The total list concatenates the coordinates of the nodes locations and their corresponding unique identifiers into one list so that it can be outputted to the user interface route log showing the user the chronological order and information about their journey.<br>Therefore each item consists of two values so the data structure can be referred to as a Tuple. |

## Validations

| **Aircraft Class Validations** |
|---|
| **Method:** Inputting wrong format and out of range variables/values into class through support variables. |
| **Description:** This form of validation is intended to confirm that no incorrect values are inputted into the aircraft variable, inputting incorrect values could cause harm to the user when the software is in use inflight, therefore the values must be accurate and in the correct format.<br>Therefore format and range limitations will be implemented in order to cleanse and validate the data and stop inaccurate data from entering the software. |

| **Fuel Consumption/Required/Cost Validations** |
|---|
| **Method:** Inputting various different variables and formats for consumption and requirements and viewing their interactions with other data structures. |
| **Description:** The validation for requirement and cost will not be able to be achieved directly as they're created from referencing and calculations but another way of validating these variables will be by inputting incorrect calculations for these variables into other data |

| |
|---|
| structures and viewing the interactions and processes that occur.<br>Therefore the validation for these variables will be implemented with format and range limitations in order to stop inaccurate data being admitted to the software that could harm the user. |

| **Nodemap Validations** |
|---|
| **Method:** Cleansing and inputting data into the subsequent functions and viewing the specific variables interactions with the node algorithm. |
| **Description:** Validation for the ‘nodemap’ variable will occur by creating a cleansed set of data by which waypoints can be formatted into lists and such, furthermore format and range checks will be admitted on the data to ensure its accuracy as well as referencing with the data given. |

## Iterative Test Data

Since the development of this software will be iterative and solutions will be implemented step by step by smaller solutions instead of one large one.
The test data will follow the same route, section by section the test data for that specific solution will be implemented and tested at that section, once the overall software is completed the whole collection of tests will be compiled to make sure there are no conflicts and that they all work cohesively.

Here’s a plan for the basis of the test data section by section where needed:

| **Test Section** | **Test Description** | **Test Data** | **Expected Outcome** |
|---|---|---|---|
| Machine Learning | Testing inaccurate and uncleansed data being inputted into the machine learning algorithm as training | Examples could include data of waypoints that do not exist, or are out of range and not | Error message explaining what test was failed and next steps. |

| | | | |
|---|---|---|---|
| | data. | possible. | |
| Route Navigation | Testing non-existent waypoints and inaccurate waypoint information to ensure the correct route is being displayed. | Non-existent waypoint codes as well as the incorrect location information for the waypoints. | Error message outputting that a non-existent waypoint has been selected. |
| Aircraft Information | Testing inaccurate and faulty information about the aircraft being inputted into the variables and classes | Data inputted into the class and subsequent variables could include the incorrect format, range or values. | An error message outputting that an incorrect variable has been inputted into the software |
| Information Output | The tests will show whether if the software is correctly appending lists and different data structure manipulations | Simply inputting nodes onto the map overlay would provide the test data needed. | Correctly appended lists should be shown in the command line, also being appropriately manipulated whether or not a node is removed from the route. |

**Test plan data examples:**

| | |
|---|---|
| Machine learning algorithm | Map input data: test coordinates (London, Birmingham, Manchester)<br>Route tests: node routes between London, Birmingham and Manchester |
| Route navigation | Placing nodes outside of route between London, Manchester and Birmingham. Such as Milton Keynes to see how it will affect the route navigation. |
| Aircraft Information | Inputting fuel consumption variables and fuel cost variables (vary depending on |

| | |
|---|---|
| | aircraft and time of operation) as well as boundary and extreme data outside the range of input values. |
| Information Output | Plotting waypoints on the overlay for the route London, Birmingham and Manchester |

## Post Development Test Data

The software after being completed will have to go through more testing to ensure that the entire testing process works cohesively with each other once the software has been completed and fully compiled.
The post development test data will be mostly based on the success criteria for the software so the overall functionality of the software is taken into account.
Therefore here is a checklist containing the test of specific functions and parts of the software that must be accounted for in the post development tests.

| **What Must Be Tested** | **Test Results** |
|---|---|
| The functionality of the start/stop journey function | |
| Graphical map overlay with accurate waypoint and route display | |
| Accurate waypoint information (latitude, longitude, altitude, unique identifier) | |
| Accurate imported map with key information | |
| Accurate representation of statistics and data from route navigation and aircraft | |
| Suitable machine learning algorithm supported by software and imported data | |

| | |
|---|---|
| sets to be trained upon | |
| Function that allows the user to view waypoint information when it is selected. | |
| Ensuring the user interface is easy to navigate around and easily accessible and understandable. | |
| Detailed user manual that accurately provides information on how to use the software and a troubleshooting guide. | |
| Provide customisation features for accessibility. | |

**How we will test each section:**

**The functionality of the start/stop journey function:**

The stop/start functionality can be implemented with a button linked to a procedure within the main class that starts and terminates the journey, in order to test this functionality routes can be created and then the button activated in order to test the responsiveness of the procedure according to its designated cause.
In terms of starting the route once again the button can be used, but alternatively the route can be cleared after the stop button and a new route created automatically after waypoints are selected into the map overlay.
To test the functionality of the start function a route can be entered and then the stop button activated and a new route can then be created, if the waypoints are represented correctly on the map overlay then the start functionality is working as expected.

**Graphical map overlay with accurate waypoint and route display:**

To ensure the overlay is properly displayed on the user interface we must test the position of overlay through trial and error using the grid method on customtktinter, this will create a grid of boxes along the GUI allowing elements to be positioned through rows and columns, we can simply place down the overlay and tweak the position to any needed adjustments.

**Accurate waypoint information (latitude, longitude, altitude, unique identifier):**

The graphical map overlay, if working correctly should display the waypoints that are being

selected in their correct positions, to test this functionality we can simply gather a list of random waypoints and their coordinates and then plot them onto the waypoint and check their corresponding overlay coordinates to their initial coordinates.
If the two sets of coordinates match up then it is confirmed that the waypoints are being accurately represented on the map overlay.
To check the accuracy of the identifiers we can simply extrapolate the list of waypoints against the list of identifiers and check each index matches with the one on the overlay.

**Accurate imported map with key information:**

The tkintermapview uses an openstreetmap tile to present the map, the openstreetmap tiles are constantly updated and will therefore allow the user to see the most accurate geographical information for their route.

**Accurate representation of statistics and data from route navigation and aircraft:**

Test data for the data representation and statistics can be acquired through either passing artificial values through a test table into the calculation formulas for the data representations or either creating test routes through the map overlays in order to retrieve the information from the route for the calculations and use that instead.
The outputs of the calculations are expected to be in a certain format and range so we can just use those expectations and apply them to the outputs of the calculations.

**Suitable machine learning algorithm supported by software and imported data sets to be trained upon:**

The machine learning algorithms can be tested using trace tables to determine their outputs and also some annotated mappings can be used to trace how the algorithm will work through data and create outputs, we can then compare these to the outputs it actually creates and then record how the algorithm performs based on its accuracy and reliability.

**Function that allows the user to view waypoint information when it is selected:**

In the map overlay the nodes can be interacted with and selected, through this the coordinate data is available, to test the accuracy of this function we will plot a couple waypoints onto the overlay and then select waypoints and then compare them with their respective real coordinates to see if they match.

**Ensuring the user interface is easy to navigate around and easily accessible and understandable:**

To test this section we will use stakeholder feedback to assess the usability and the

functionality of the software and how it performs, through this feedback we can gain insights on user preferences and how to further optimise the software for users.

**Detailed user manual that accurately provides information on how to use the software and a troubleshooting guide:**

This section can also be tested through user feedback and their responses, ideally for this section of testing the outcome should be that the user feels as if the manual helps them learn about how to use the software and do not have to enquire any further.

# Development And Testing

## Graphical User Interface Development

**Stage 1 Development: Graphical User Interface**

For the Active Route Finders graphical user interface the Python 'tkinter' and 'customtkinter' module will be used in order to provide an user interface between the user and the software, the modules will allow a seamless integration between the functionality and accessibility of the program by providing GUI aspects such as overlays and entry fields which are connected to backend functions that will return feedback to the user.

tkintermap2.py

```
1  import tkinter as tk
2  from tkinter import *
3  import customtkinter as ctk
4  import tkintermapview as tmv
```

The main 3 import consist of the tkinter, customtkinter and tkintermapview modules, these modules allow an interactive GUI to be formed and the tkintermapview module allows an openstreetmap map tile to be placed into a tkinter widget allowing a map overlay to be accessible and viewable by the user in the window itself without having to rely on HTML pages to output map displays.
Functions such as the inputting of location information will be linked to GUI aspects and widgets, allowing cohesive action between the backend and the frontend.

```
6  class arf(ctk.CTk):
7      def __init__(self):
8          super().__init__()
9
10         self.title('active route finder')
11         self.geometry('1700,900')
12         self.minsize(1700,900)
13
14         self.mainloop()
15
16 arf()
```

The class arf will contain the code for displaying the window itself to the users device, the class parameters are defined at ‘ctk.CTk’ which displays the window and then the .title, .geometry and .minsize methods are used to manipulate different aspects of the window allowing for customization, in the future the methods could be set up in a way so that a tkinter widget can allow the user to change the inputs to the methods for increased accessibility.
The output of the window is defined by the ‘.mainloop()’ method, this ceases and functions after this line and also allows interrupts to interact with the program and constantly updates the window that's being outputted to the user.
On the other hand the dunder init method is a constructor which is used to initialise the attributes of the object which would be the arf class.

This is the output produced by the code above :


These three components of the tkinter module will house the buttons, widgets, frames and functions to cooperate with each other to create a final accessible user interface.
In the case of Active Route Finder the tkinter and customtkinter framework will be implemented within functions and classes, which will therefore allow easier maintainability of the software as well as an easier and more streamlined development process as most of the processes will be contained within the classes.

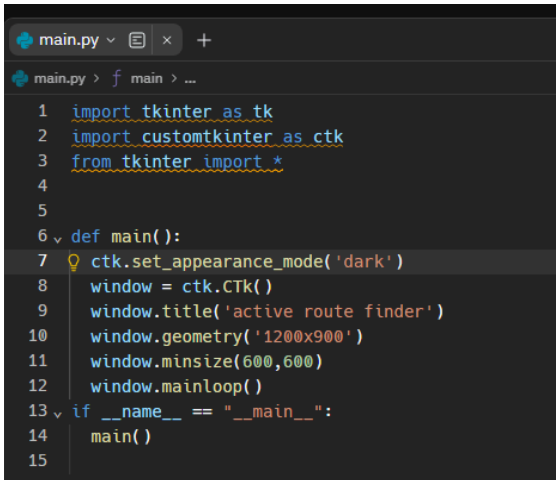
```
1  import tkinter as tk
2  from tkinter import ttk
3  import customtkinter as ctk
4
5  ctk.set_appearance_mode('Dark')
6
7  class arf(ctk.CTk):
8      def __init__(self):
9          super().__init__()
10         self.title('Active Route Finder')
11         self.geometry('800x750')
12         self.minsize(400,400)
13
14
15
16         self.mainloop()
17
18 arf()
```

The basis of the active route finder GUI will be created using a class method named ‘arf’, through this method the customtkinter library is inputted as a parameter, and then the dunder init method is used as it initialises the custom tkinter inside the object which will then be embedded inside the ‘arf’ class.
Furthermore the ‘super’ method is also used on line 9 in order to initialise the attributes of the parent class which in this case would be ‘arf’.

```
class menu(ctk.CTkFrame):
    def __init__(self, parent):
        super().__init__(parent)
```

A second class called ‘menu’ was created to store the widgets for the GUI inside an object, here the dunder init and super dunder init methods are used with the parameter ‘parent’ as a master in order to call the object ‘ctk.CTkFrame’ as if it is itself.

With the tkinter library to display the map overview, tkintermapview will be integrated, this allows a openstreetmap overlay to be displayed within a tkinter widget.




Here's another form of displaying the customtkinter window which will contain the widgets for the application.
The loop for displaying the window to the screen will be contained in a procedure called 'main', the purpose of line 13 and 14 are to make sure that the specific block of code in the procedure is executed, this will become necessary in the future as more and more classes, functions and procedures will be implemented into the code.
Controlling what blocks of code are executed at what time can improve maintainability and reduce the complexity of the code making it easier to run and understand from a developer viewpoint.

```
61 class layout(ctk.CTkFrame):
62     def __init__(self, parent):
63         super().__init__(parent)
64
65
66         self.place(relx=0.3, y=0, relwidth=0.7, relheight=1)
67
68         overlay = tmv.TkinterMapView(self, width=900, height=600)
69         overlay.place(relx=0.13, rely=0.16)
70         overlay.set_zoom(15)
71
72         marker = overlay.set_position(51.5228719, -0.0094993, marker=True, text_color='black', font='calibri')
73
74         marker1 = overlay.set_position(51.523679, -0.072939, marker=True, text_color='black', font='calibri')
75
76         marker.set_text('bromley-by-bow')
77         marker1.set_text('shoreditch')
78         print(marker.position, marker.text)
79         print(marker1.position, marker1.text)
80
81         path = overlay.set_path([marker.position, marker1.position, (51.5228719, -0.0094993), (51.523679, -0.072939)])
```

This class ‘layout’ created a frame via from the customtkinter library and places it onto the window, through the ‘tkintermapview’ library an overlay of an openstreetmap is created and attributes can be attached as methods such as .set_position, .set_text and .place.
The map overlay is created through an openstreetmap tile and is placed onto the frame, the width and height of the overlay can be changed through commands in the attributes.
For future reference it may be useful to add some f’ strings to allow the user to input a value and have the results change the window.

In terms of map positioning and nodes there are commands to input coordinates and addresses into .set_position, which when compared with the marker command can display a pinpoint on the coordinates location. For future reference an entry widget could be created in customtkinter that allows the user to enter a set of coordinates for their waypoint.
Through this nodes will be displayed at the users request onto the overlay for use.
On the other hand the .set_path method joins together two nodes/coordinates and creates a line between them, this path will be used to connect nodes and create a route for the user to follow.
In terms of user input, entry fields and such will be available from frames that will be linked to functions and be opened when a certain button is interacted with.

**Issues with frame switching and solution:**

The initial plan was to use customtkinter in order to link buttons on the main menu to different frames that could be raised and therefore show different information to the user depending on the purpose and events, however raising frames in customtkinter has proven to be more difficult than raising and switching frames in tkinter, perhaps due to a lack of documentation for customtkinter and more people using tkinter for their graphical user interfaces, the idea to use frame switching was scrapped as the software was planned to be coded exclusively in customtkinter for styling and simplicity purposes.

Research was concluded and a new method to show different widgets on the same window was found, customtkinters tab view method allows a tab of different views to be placed onto the main menu and then the user can press on a specific tab, concurrent to their needs, the tab pressed will display different widgets onto the same window that the program was launched with so no customtkinter top level methods will be used to display new menus.
However the customtkinter top level can be combined with the tab view in order to allow the user to input data and information necessary to the program into the software.

```python
import customtkinter as ctk
import tkintermapview as tmv

#presetting window theme and mode

ctk.set_default_color_theme('blue')
ctk.set_appearance_mode('dark')

class arf(ctk.CTk):
    def __init__(self, title, size):
        super().__init__()

        #creating presets for window size and title

        self.title(title)
        self.minsize(size[0], size[1])
        self.maxsize(size[0], size[1])
        self.geometry(f'{size[0]}x{size[1]}')
```

To improve the simplicity of the program the parameters in the methods were switched to have arrays from the mainloop function being fed instead of manually typing it into the geometry and title areas, therefore to maintain the title and geometry of the window only one line has to be edited now instead of 2.

```python
if __name__ == '__main__':
    app = arf('active route finder', (1500,900))
    app.mainloop()
```

This script will run at the end of the file and will be where the title and window geometry are inputted, I have also included on line 99 the code that ensures that the main file is executed which will be the file containing all of the graphical user interface code, after this if requirement is met then the applications mainloop will activate.

**This is the initial iteration of the graphical user interface code:**

```
20          #creating grid for widgets
21
22          self.grid_rowconfigure((0,1,2), weight=1)
23          self.grid_columnconfigure(1, weight=1)
24          self.grid_columnconfigure((2,3), weight=0)
25
26          #creating frame for menu buttons
27
28          self.barFrame = ctk.CTkFrame(self, width=100, corner_radius=10)
29          self.barFrame.grid(row=0, column=0, rowspan=4, padx=10, pady=10, sticky='nsew')
30          self.barFrame.grid_rowconfigure(4, weight=1)
31
32          #adding widgets to frame
33
34          self.text1 = ctk.CTkLabel(self.barFrame, text='active route finder')
35          self.text1.grid(row=0, column=0, padx=10, pady=10)
36
37          self.button1 = ctk.CTkButton(self.barFrame, text = 'start/stop', command = lambda: print('button1 pressed'))
38          self.button1.grid(row=1, column=0, padx=10, pady=10)
39
40          self.button2 = ctk.CTkButton(self.barFrame, text='input data', command = lambda: print('button2 pressed'))
41          self.button2.grid(row=2, column=0, padx=10, pady=10)
42
43          self.button3 = ctk.CTkButton(self.barFrame, text='user manual', command = lambda: print('button3 pressed'))
44          self.button3.grid(row=3, column=0, padx=10, pady=10)
45
46          self.appearanceModeText = ctk.CTkLabel(self.barFrame, text='appearance', anchor='w')
47          self.appearanceModeText.grid(row=5, column=0, padx=10, pady=10)
48          self.appearanceModeOption = ctk.CTkOptionMenu(self.barFrame, values=['light', 'dark'], command=self.changemode)
49          self.appearanceModeOption.grid(row=6, column=0, padx=10, pady=10)
```

Comments have been added along with whitespace in order to improve developer readability and make it easier to maintain the code.
The code continues from the arf class, the class allows easier asset management, therefore improving maintainability as an issue can be contained within a class, this also makes it easier to have certain components cooperate with each other.

Lines 22 to 24 set the grid and row configure for the window, I have chosen to create 2 rows with a weight of 1 which means that it is offset from the centre along with one column, this will create the bar for widgets along the side which has been moved from the bottom to the left side in order to improve accessibility and usability of the software for the user.
A second grid is then configured with 2 columns stretching to the top and the bottom of the window area.
Furthermore along with every button is a lambda command that will print that the button has been pressed should it be interacted with, this will help in development to make sure that buttons are registering inputs from the user, and to create a log of user interactions.

Lines 28 to 30 create a frame within the grid so that widgets are able to be laid onto the window for the user to interact with, frames can allow different components of the window to be separated from each other in order to stop any widgets from clashing with each other during usage.

**Dedicated frame for tkintermapview:**

```
51          self.displayFrame = ctk.CTkFrame(self, width=1000, corner_radius=10)
52          self.displayFrame.grid(row=0, column=1, rowspan=2, columnspan=1, padx=10, pady=10, sticky='nsw')
53          self.displayFrame.grid_rowconfigure(2, weight=1)
```

The display frame will contain the tkintermapview overlay, there has been a preset width assigned to the frame so the map won't experience any scaling issues.
A grid for the row config has been set as well inside the main barFrame allowing the map widget to be positioned in the window accurately, furthermore the use of the sticky command has been used to ensure that the widget will take up all available space on the north, south and west directions, but not east as that will contain other widgets.

**Changes to tkintermapview script:**

```
55          self.overlay = tmv.TkinterMapView(self.displayFrame, width=900, height=600)
56          self.marker = self.overlay.set_position(51.522, -0.009, marker=True, text_color='black', font='calibri')
57          self.marker.set_text('bow')
58          print(self.marker.position, self.marker.text)
59          self.overlay.place()
60          self.overlay.grid(row=0, column=1, padx=10, pady=10 )
```

The tkintermapview module and its functions have now been embedded into the class therefore making it so that the output from the tkintermapview module stays within the display frame.
The main changes that have been made to the script is that the lines have simply been prefixed with self. And the frame has been changed to self.displayFrame, the rest of the code has been kept fundamentally the same.
This can be expanded to link with widgets to input addresses and coordinates in order to be transferred directly onto the tkintermapview display and outputted to the user.

However one possible issue that could arise with this method is the loss of modularity within the active route finder scripts as instead of a traditional setup where multiple classes would be linked together, most of the functionality is included into the main arf class along with some functions and procedures in order to supplement the commands that will be linked to the buttons and widgets in the main frames for the user interactions.

**Maintenance carried out so far:**

So far there have been issues with the .grid() methods on placing widgets across the frames within the window, errors have occurred where widgets aren't positioned correctly within their frames or are offset and exceeding and stretching the frame, causing conflicts with other frames on the window.
One specific issue with the grid method that occurred was the positioning of the tkintermapview overlay on the displayFrame, when positioned initially the overlay was not taking up the full width of the frame on the window, instead only taking up a portion of the displayFrame and not responding to commands with row, column, rowspan and columnspan. As well as issues when placing objects into frames causing them to expand and distort over the original areas leading to a dishevelled appearance to the user interface.
Issues were also occurring where another widget was added to a frame with existing widgets which then caused the pre-existing widgets to move out of the frame and into the overlay frame where the mapping is housed.

In order to try and solve this I tried to implement propagate commands into the program on the frame attributes in the program, the propagate command ensures that a frame will not change its size in order to wrap around a widget, therefore remaining consistent as new widgets are added to the frames.

**Creation of the widgets for the user interface**

One the graphical user interface there will be a collection of two frames, one of which will contain the map overlay for the user and the other frame will contain the widgets for the user input and functionality for the program.
Below is the code for the tkintermapview overlay, some test locations and data have been inputted as parameters in order to display a marker, eventually the user will input multiple markers from their own geographical data in order to create a path that they can follow and use as their route and for organisation within the program.

As well as this the .grid_propogate command has been included to make sure there's no resizing of the frame in order to cause the frame to wrap around the widgets size and therefore cause sizing and scaling issues.

```
64
65          #create a frame for the map overlay
66
67          self.displayFrame = ctk.CTkFrame(self, corner_radius=10)
68          self.displayFrame.grid(row=0, column=1, columnspan=1, rowspan=3, padx=10, pady=10,
    sticky='nsew')
69          self.displayFrame.grid_columnconfigure(1, weight=1)
70          self.displayFrame.grid_rowconfigure(1, weight=1)
71          self.displayFrame.grid_propagate(False)
72
73
74          #add overlay to overlay frame
75
76          self.overlay = tmv.TkinterMapView(self.displayFrame)
77          self.mapInfo = self.overlay.set_position(51.522, -0.009, marker=True,
    text_color='black', font='calibri')
78          self.mapInfo.set_text('bow')
79          self.overlay.place()
80          self.overlay.grid(column=1, row=1, columnspan=1, rowspan=1, padx=10, pady=10,
    sticky='nsew')
81
82          #printing the position data for testing reasons
83
84          print(self.mapInfo.position, self.mapInfo.text)
```

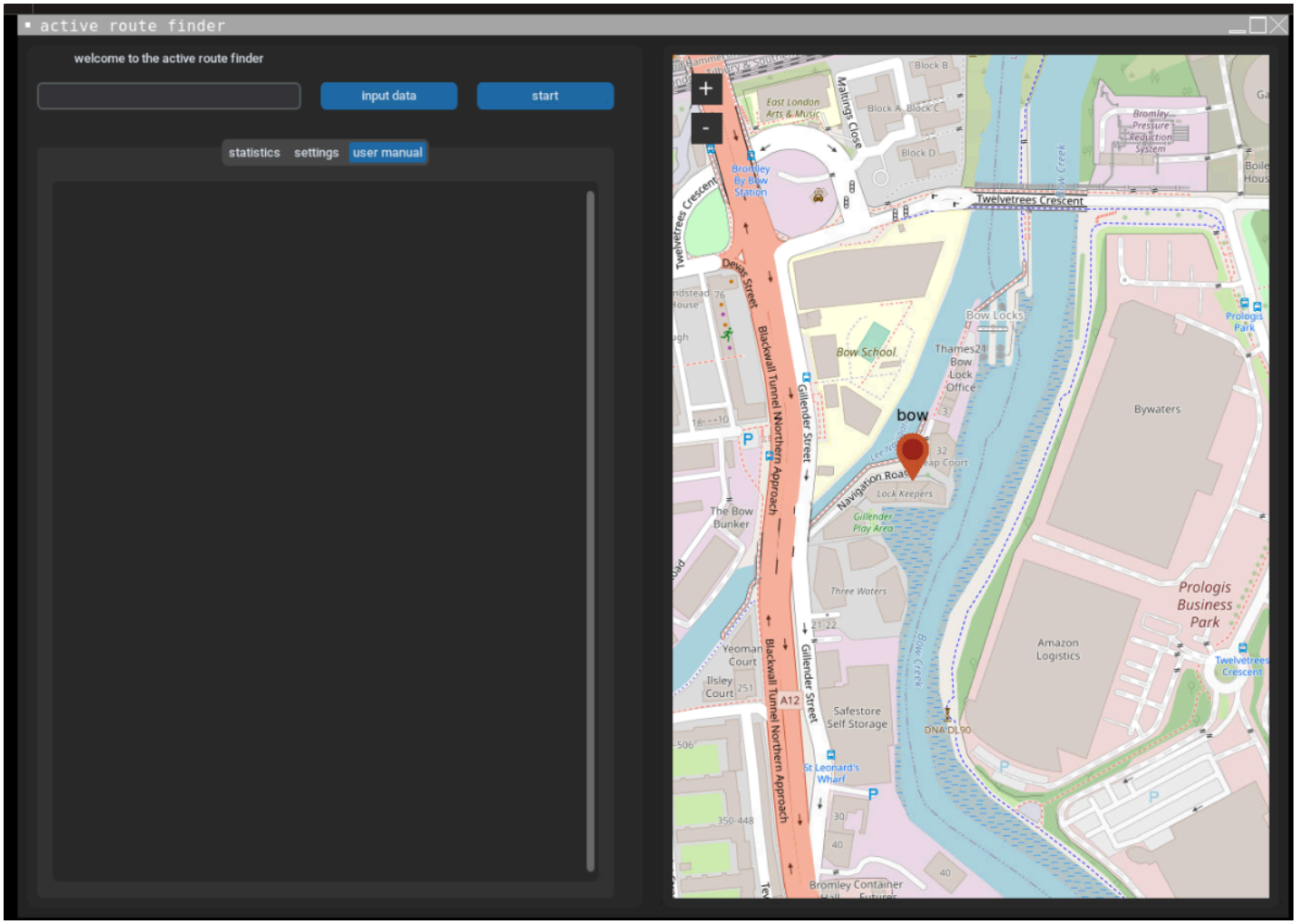
Line 84 is used as test data and in this case will output:

```
python3 main.py
(51.522, -0.009) bow
```

This can be inputted into an array and stored in a route log which can help with statistics and further testing for the active route finder.
These coordinates can be passed through into a textbox which can display a log of the nodes and their locations respectively in order.

Below is the output for the combined frames of the functionality and display sections of the program, on the right hand side the tkintermapview module is presenting its openstreetmap tile onto the customtkinter frame.
There's also a marker displayed in the map attributes that can be given to the marker such as colour, text colour, name and different icons.

On the other side the widgets for inputting information and submitting it is included, furthermore there's a tkinter tabview module which will house more frames such as the user manual, settings, stats page and in the future the route log.
The frames for functionality and mapping are split down the middle of the window in order to produce an ergonomic and easy to use graphical user interface.




**Development phase for the GUI structure has now completed:**

I've successfully completed the structure for GUI, and added many features to improve accessibility. Now the development phase for the tabs on the screen will start. Furthermore a number of the post development test criteria have been met.

**Code for the user manual text box:**

```
87        #create and add widgets to the usermanual tab
88
89        self.scrollable = ctk.CTkScrollableFrame(self.multiBox.tab('user manual'))
90        self.scrollable.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
91        self.scrollable.grid_columnconfigure(1, weight=1)
92        self.scrollable.grid_rowconfigure(1, weight=1)
93
94        self.manualText = ctk.CTkTextbox(self.scrollable, corner_radius=10)
95        self.manualText.grid(column=1, row=1, columnspan=1, padx=10, pady=10, sticky='nsew')
96        self.manualText.configure(state='disabled')
97
98        self.manualText.insert('0.0', 'the active route finder is a tool aimed at private pilots.')
```

Here a customtkinter scrollable frame has a textbox put inside it, the scrollable frame was given a one row and one column grid to place the textbox which has an aim of being able to use the scrollable frame in order to scroll through the text displayed on the textbox.
However when the code was run this was the result.

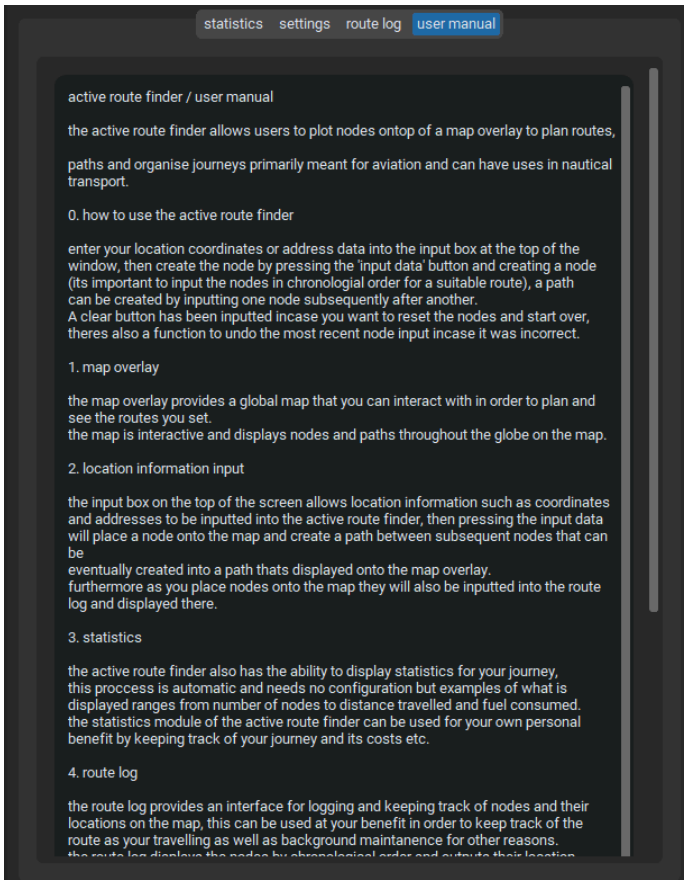

The textbox was not taking up the whole of the scrollable frame leading to large amounts of white space that has no purpose, in order to fix this, this code was deployed.
As a result the scrollable frame was serving no purpose in this context as the text box simply did not reach down far enough in order for a scrollable frame to even become active, therefore some changes had to be made in order to firstly make sure the text box takes up all

available space on the frame and secondly to make sure that the text can be read from a text file.

**Here's an updated look at the code:**

```
        self.scrollable = ctk.CTkScrollableFrame(self.multiBox.tab('user manual'))
        self.scrollable.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
        self.scrollable.grid_columnconfigure(1, weight=1)
        self.scrollable.grid_rowconfigure(1, weight=1)

        self.manualText = ctk.CTkTextbox(self.scrollable, corner_radius=10, height=1000)
        self.manualText.grid(column=1, row=1, columnspan=1, padx=10, pady=10, sticky='nsew')
        self.manualText.configure(state='normal')

        f = open('arfusermanual.txt', 'r')
        textFile = f.read()
        f.close()

        self.manualText.insert('0.0', textFile)
```

**Here’s the updated output:**




**Iterative update on the user manual text tab:**

Issues were occurring in the graphical user interface where in the user manual tab there were two scrollbars showing, one for the scrollable frame and one for the text box itself, therefore I decided to change the method to show the text onto the scrollable frame, instead now I'll be using the customtkinter label function in order to display the user manual text.
Some benefits that come with this is that the user will not be able to accidentally edit the text as there was no way to disable edits being made to the text as it was fundamentally a text box widget.
Therefore for the text box widget a line of code shown below had to be added to change the state of the text box to 'normal' in order to allow it to function.

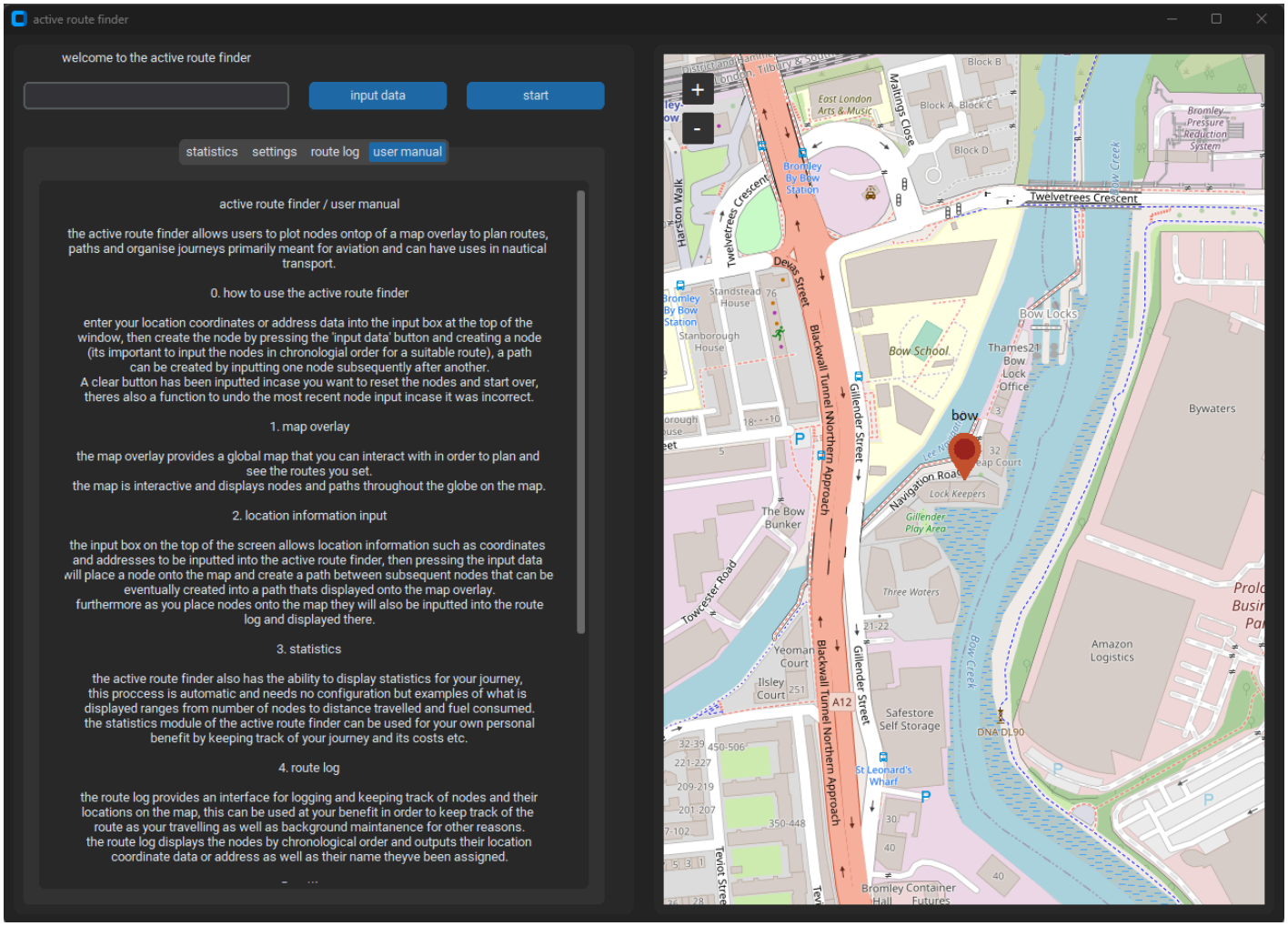
```
102            self.manualText.configure(state='normal')
```

The new and updated code can be seen below:

```
91          #create and add widgets to the usermanual tab
92
93          self.scrollable = ctk.CTkScrollableFrame(self.multiBox.tab('user manual'))
94          self.scrollable.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
95          self.scrollable.grid_columnconfigure(1, weight=1)
96          self.scrollable.grid_rowconfigure(1, weight=1)
97
98          self.manualText = ctk.CTkLabel(self.scrollable, corner_radius=10, height=1000)
99          self.manualText.grid(column=1, row=1, columnspan=1, padx=10, pady=10, sticky='nsew')
100
101
102         #lines 104 to 108 open an external text file in order to read the user manual and place it into the user manual tab
103
104         f = open('arfusermanual.txt', 'r')
105         textFile = f.read()
106         f.close()
107
108         self.manualText.configure(text=textFile)
```

Furthermore the read file text at line 108 was also changed to allow the text file to be read to the label.
As a result due to this change the code has improved its simplicity and maintainability by first of all reducing the amount of code to output the same function but also second, improving the simplicity and looks of the graphical user interface.

An updated view of the user manual tab can be seen below:




A text file has been created with the user manual, the open and read commands were then used in order to open the text file containing the instructions and information, the commands can be seen on line 98 to 100 and to make sure that the commands were carried out correctly the .close() command was included.
I've decided to use a customtkinter text box instead of a label as it can be integrated with the customtkinter scrollable frame in order to create a more accessible program for the user.

The next step of production is to fix and create a more usable settings tab on the functionality and user input frame, so far there's very limited options to what the user can customise.
Initial ideas are to add a scaling option as well as optimising the positioning of the theme buttons for the window.
Scaling will be possible by creating a function outside the arf class functions inside the function it will contain commands for picking different scale values and then activate the selected scale factor and deploy it into the application appropriately.

**Here's the updated code for the scaling menu:**

```
87      self.scalingMenu = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values=['100%', '80', '120'], command = self.differScale)
88      self.scalingMenu.grid(row=3, column=0, padx=5, pady=5)
89
```

```
136
137     def changemode(self, newmode: str):
138       ctk.set_appearance_mode(newmode)
139
140     def differScale(self, newScale: str):
141       newScale_float = int(newScale.replace('%', '')) / 100
142       ctk.set_widget_scaling(newScale_float)
143
```

The number of lines has successfully been decreased therefore allowing for easier maintainability.

Here’s the code for the route log tab:

```
110     #create frame for the route log tab
111
112     self.scrollableLog = ctk.CTkScrollableFrame(self.multiBox.tab('route log'))
113     self.scrollableLog.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
114
115     self.logText = ctk.CTkLabel(self.scrollableLog, corner_radius=10, height=700)
116     self.logText.grid(row=1, column=1, columnspan=1, rowspan=1, padx=10, pady=10, sticky='nsew')
117
118     #reading from the external file that contains arrays of the node information
119
120     r = open('arfrouteinfo.txt', 'r')
121     routeFile = r.read()
122     r.close()
123
124     self.logText.configure(text=routeFile)
```

Similar enough to the user manual tab, text is being read from the text file ‘arfrouteinfo’, the functionality for the file has not been completed yet but aims are to make sure that the information inputted by the user into the input bar at the top of the window is taken and then stored into the text file as an array (most likely to be two dimensional), after this the array will be read from the text file and outputted onto the route log scrollable frame.

This loop of the writing to the text file and then reading will provide a secure form of storage for the node data and then reading it into the route log in order to provide the data of all the placed nodes for the users benefits, this could also come in helpful in maintaining the system as well as inputting test data.

The routing information will be displayed onto a customtkinter label on the route log tab, the data will be outputted in a format similar to this,


The bracket will contain either coordinate data or address information, and outside the bracket will include the node name, aims are to create unique node identifiers for each one through a random 4 letter code generator or by other means, the node identifier 'bow' was a test to insure that the node marker and identifier was showing correctly on the tkintermapview overlay.

**Development phase for customtkinter entry bar functionality:**

The idea for the entry bar is to allow the user to input data for their nodes through address or coordinate data, functionality behind the input bar will include taking the input and storing it within a text file as an array or as a list, from there functionality will be connected to the route log so when the user accesses it they can see the nodes placed so far and all of the information.

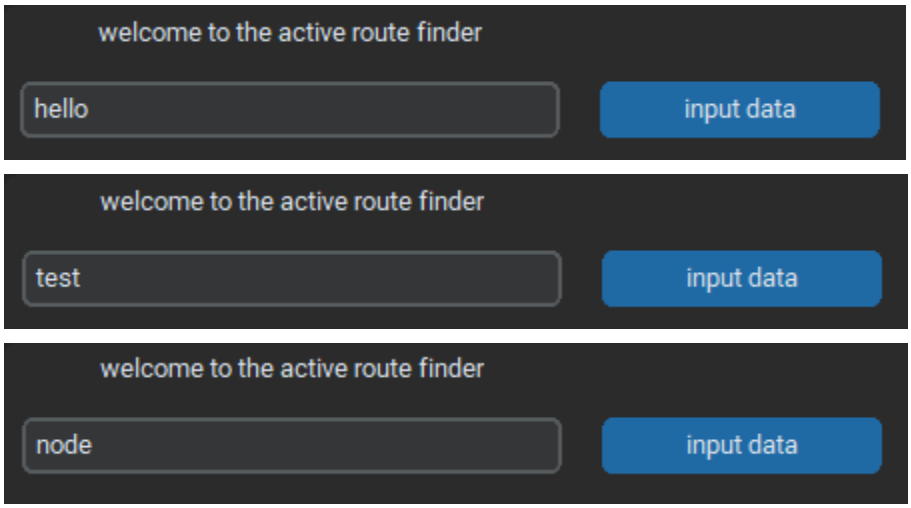
```
170     def userInput(self):
171         global nodeData
172         nodeData = self.entryBox.get()
173 
174         r=open('arfrouteinfo.txt', 'a')
175         r.write(nodeData)
176         r.close()
177         print(nodeData)
```

I created this subroutine userInput in order to take data inputted from the entry box to the text file, the append command was used on the file in order to add strings to the text file, while the .get() function was used on the entry box itself in order to retrieve the information the users inputted into the entry bar in correspondence to the route they would like to plot.
To ensure the reusability of the procedure the variable nodeData was included into the global scope for variables in order to be used in the functionality for the entry field in the future.

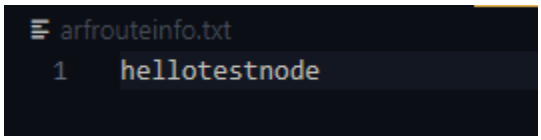
```
self.entryButton = ctk.CTkButton(self.barFrame, text='input data', command = self.userInput)
```

Simply appending the command to the entry button allows the subroutine to be activated when the user presses the button in order to input data, further on I would like to implement the function to clear the entry bar once the entry button has been pressed in order to improve convenience so that the user does not have to manually clear the input bar after every input.

**Testing the entry box functionality:**




These three pieces of test data were inputted into the entry bar, and then the input data button was pressed after, ideally the data should go through the entry bar into the subroutine and be outputted into the console as well as the text file so let's check the results.

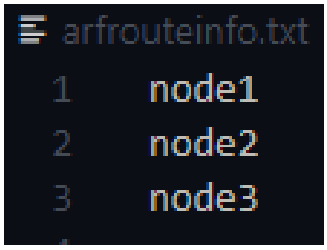



The strings were inputted into the text file however they were not formatted correctly therefore causing the strings to concatenate together on one line, this will cause issues when retrieving the strings or real values again for display in the route log.

**Fixing the issue:**

```
    def userInput(self):
        global nodeData
        nodeData = self.entryBox.get()

        r=open('arfrouteinfo.txt', 'a')
        r.write(nodeData)
        r.write('\n')
        r.close()
        print(nodeData)

    def clearInput(self):
        self.entryBox.delete(0, 100)

if __name__ == '__main__':
    app = arf('active route finder', (1300,900))
    app.mainloop()

print('user has exited the active route finder')

with open('arfrouteinfo.txt', 'r+') as dataFile:
    dataFile.truncate(0)
```

Through using the /n character, every subsequent input will not be printed onto a new line in the text file, the new output looks like this.




Now every subsequent input is printed onto a new line readability for the route log will be improved, furthermore the truncate method was included after the mainloop in order to clear the route info file after the program is terminated by the user.
This was done so that it would be more convenient as every time the user starts up the active route finder in order to plot a new route there would be a blank data file for the route log and therefore making the program more coherent with every single use.

In order to transfer the inputs from the input box to the route log a simple .get() function can be used in order to store the data from the entry box and then transfer it to the route log label where it will be displayed.
Furthermore the node data has to be stored in order to be transferred to the statistics tab as well, where data about the nodes will be compiled and then displayed in graphical form.

For the data storage of the node data in a text file, the data instead of being stored freely line by line could be stored in an array, tuple, dictionary, list or stack or any of those data structures, some benefits of using these data structures as storage methods is that they may improve the efficiency of the program from reading the node data from the text and transferring it to the multiple places.
In the future this will help with performance as well as the processes running on the users device, so maintaining efficiency is important to make sure that the active route finder does not crash whilst the user is using it.

**Iterative development for the statistics tab:**

First of all the prototype for the matplotlib integration with customtkinter must be created in order to ensure that the two modules can work together cooperatively without any conflicts and to plan out styling and functionality

To be able to implement matplotlib into customtkinter I will firstly be using matplotlib.pyplot which allows for interaction between the user and the plots, furthermore I will be using matplotlib backend classes such as FigureCanvasBase to separate the figure from the user interface area in order to create modularity.
Also another backend class I will be using is backend_tkagg which will aid in the placing of plots onto the customtkinter graphical user interface.
Finally the numpy module will be implemented in order to format data so it can be inputted into the plots and eventually displayed to the graphical user interface.

## Stage 1: Graphical User Interface Review

In the first stage the core graphical user interface was created using object oriented programming techniques, as well as modularity via classes and subroutines.
Using these methods I was able to create a simple but modern GUI that the user should have no issue interacting with, the GUI has 4 tabs on the left, the data tab, the settings tab, the route log tab and the user manual tab, while on the right the map overlay is displayed.

So far the graphical user interface has been tested through observation, by making sure that widgets do not visibly clip or overlap each and that frames have False propagation values in order to ensure that they won't expand or shrink to fit the widgets inside them, as well as by using trial and error to place widgets using the .grid() method.

In terms of meeting the success criteria and user expectations, one of the major criterias was an easy to use and modern interface that increases the user experience while still staying functional, I have included an openstreetmap through the module tkintermapview in order to display an up to date global map that will eventually display accurate waypoints.
Furthermore a user manual and data section were added in order to appease user expectations by improving the user experience.

A few changes have been made from the design phase, which the largest one would be the layout of the GUI itself, it has transformed to make a more modern appearance as well as boasting new features and options to improve customizability of the experience

A summary of the prototype as a whole at this stage would conclude that once the functionality starts being added to the GUI, then the coherence of the concept will start to make more sense and also at this moment lots of room has been created for future expansion and maintenance of the active route finder.

**Defined Imports:**

```
import haversine as hs
import datetime as dt
import numpy as np
import customtkinter as ctk
```

In order to make sure that calculations such as fuel consumption are calculated correctly and accurately, some information is going to have to be inputted into the entry boxes by the user. From here the get() method can input the values into the formulas in order to calculate and output the information for the user.

In order to calculate the fuel consumption we are going to have to calculate the distance between the coordinates, hence this is why the inputs are gonna be limited to integers and floats so we can use a formula to calculate the distance between two coordinates taking into account the Earth's shape (as it is a sphere).

The Haversine formula is what I will be using, it calculates the distance between coordinates on a sphere in order to be accurate compared to distances on a flat plane.
Using some trigonometry formulas the distance can be calculated with ease.

Here's the Haversine formula:

$$a = \sin^2(\Delta\varphi/2) + \cos\varphi 1 \cdot \cos\varphi 2 \cdot \sin^2(\Delta\lambda/2)$$

$$c = 2 * \operatorname{atan2}(\sqrt{a}, \sqrt{(1-a)})$$

$$d = R * c$$

Where φ represent the latitudes, and λ represent the longitudes.

In order to simplify the script instead of coding out the Haversine formula itself, where errors could easily occur we can instead import the haversine library which allows coordinates to be inputted into a function and the distance to be outputted which can have its units changed to even nautical miles, this will improve the accessibility of the active route finder.

```
bow = (51.527792, -0.011439)
manchester = (53.481125, -2.222593)

#haversine calculation in kilometres
h1 = haversine(bow, manchester)

#haversine calculation in miles
h2 = haversine(bow, manchester, unit='mi')

#haversine calculation in nautical miles
h3 = haversine(bow, manchester, unit=Unit.NAUTICAL_MILES)

print(h1, h2, h3)


print('calculations terminated')
```

Here is a prototyped test for the haversine module where on the first couple of lines coordinate data can be inputted into a variable, in the future the data can be inputted through a customtkinter entry box where the get() method can then be used for the integration with customtkinter within the future.

The haversine module allows for different units to be used in the calculation for the distance such as kilometres, miles and nautical miles.

The output for the print statement outputs this:

```
' '--' 'C:\Users\student\Documents\activeroutefinder\distance.py'
263.74157413260946 163.88141629873593 142.40905718683132
calculations terminated
```

The issues here is that the output of the distances is too large can be confusing for users to read and also will produce longer calculations with answers that will have many decimal places, in order to fix this we will use python's rounding function in order to round the coordinates to 2 decimal places in order to keep accuracy but also keep the data cleansed enough to create a valid output for the users and subsequent calculations.

The new rounded script and output looks like this:

```
bow = (51.527792, -0.011439)
manchester = (53.481125, -2.222593)

#haversine calculation in kilometres
h1 = haversine(bow, manchester)
h1r = round(h1, 2)

#haversine calculation in miles
h2 = haversine(bow, manchester, unit='mi')
h2r = round(h2, 2)

#haversine calculation in nautical miles
h3 = haversine(bow, manchester, unit=Unit.NAUTICAL_MILES)
h3r = round(h3, 2)

print(h1r, h2r, h3r)


print('calculations terminated')
```

```
' '--' 'C:\Users\student\Documents\activeroutefinder\distance.py'
263.74 163.88 142.41
calculations terminated
PS C:\Users\student\Documents\activeroutefinder>
```

Now the outputs for the distances calculated are rounded to two decimal places.

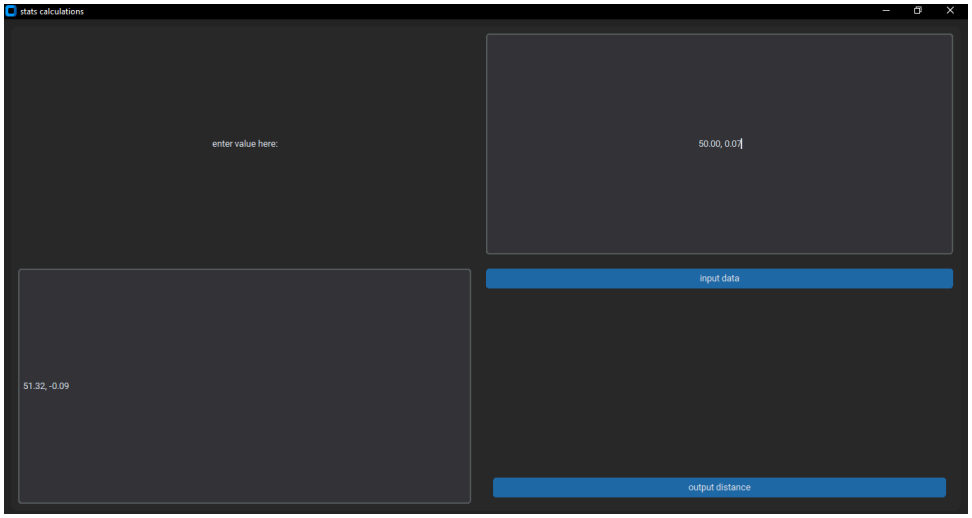
**Here is the code for the updated object oriented script:**

For the code object oriented programming and modularity have been implemented in the form of classes and procedures in order to improve the efficiency of the code itself

```
class calculations(ctk.CTk):
    def __init__(self):
        super().__init__()

        self.grid_columnconfigure((0,1), weight=1)
        self.grid_rowconfigure((0,1), weight=1)

        self.Frame = ctk.CTkFrame(self, corner_radius=10)
        self.Frame.grid(row=0, column=0, rowspan=2, columnspan=2, padx=10, pady=10, sticky='nsew')

        self.Frame.grid_propagate(False)

        self.Frame.grid_columnconfigure((0,1), weight=1)
        self.Frame.grid_rowconfigure((0,1), weight=1)

        self.geometry('1000x800')
        self.title('stats calculations')

        self.entry1 = ctk.CTkEntry(self.Frame, justify='left')
        self.entry1.grid(row=1, column=0, padx=(10,10), pady=(10,10),  sticky='nsew')

        self.entryLabel = ctk.CTkLabel(self.Frame, text='enter value here:')
        self.entryLabel.grid(row=0, column=0, padx=(10,10), pady=(10,10), sticky='nsew')

        self.entryButton = ctk.CTkButton(self.Frame, text='input data', command = self.input1)
        self.entryButton.grid(row=1, column=1, padx=10, pady=10, sticky='new')

        self.entry2 = ctk.CTkEntry(self.Frame, justify='left')
        self.entry2.grid(row=0, column=1, padx=10, pady=10, sticky='nsew')

        self.printButton = ctk.CTkButton(self.Frame, text='output distance', command = self.haversineformula)
        self.printButton.grid(row=1, column=1, padx=20, pady=20, sticky='sew')
```

```
    def input1(self):
        global node1 ; node1 = self.entry1.get()
        global node2 ; node2 = self.entry2.get()

    def haversineformula(self):
        global distance ; distance = haversine(node1, node2)
        print('distance is ', distance)

if __name__ == '__main__':
    app = calculations()
    app.mainloop()

print('calculations terminated')
```

**Here is the output for the script:**




**Error in the script: (Test for haversine formula)**

However once the script was run some errors were found concluding to the formatting of the data that was inputted to the haversine formula.

```
Exception in Tkinter callback
Traceback (most recent call last):
  File "C:\Program Files\Python312\Lib\tkinter\__init__.py", line 1948, in __call__
    return self.func(*args)
           ^^^^^^^^^^^^^^^^
  File "C:\Users\student\AppData\Roaming\Python\Python312\site-packages\customtkinter\windows\widgets\ctk_button.py", line 554, in _clicked
    self._command()
  File "C:\Users\student\Documents\activeroutefinder\distance.py", line 47, in haversineformula
    global distance ; distance = haversine(node1, node2)
                                 ^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\student\AppData\Roaming\Python\Python312\site-packages\haversine\haversine.py", line 210, in haversine
    lat1, lng1 = point1
    ^^^^^^^^^^
ValueError: too many values to unpack (expected 2)
```

Reading the error statement it seems that there is an error in the way that Python reads the list input by the user and then assigns it to the variables node1 and node2, this occurs as the number of variables is not currently matching the number of inputs given.
As a result the error is raised, in order to solve this issue we are going to have to individually assign every input to the variable in order to ensure that Python can unpack the inputs properly and assign them to either variables node1 or node2.

**Solution to issue:**

As it turns out the haversine module in python was the issue, as it takes variables into the haversine function as unpacked variables which usually have to be float values, as a result in order to keep the code efficient and simple I changed the distance calculating module to geopy which has a distance function that does the same exact execution as haversine including different measuring units.

**Test code:**

```python
from geopy import distance

node1 = input('input node1 coords: ')
node2 = input('input node2 coords: ')


d = distance.distance(node1, node2).miles

dr = round(d, 2)
print(dr)
```

```
hon312\python.exe' 'c:\Users\student\.
input node1 coords: 51.7535,-0.0967
input node2 coords: 51.2921,0.0424
32.46
```

As you can see the nodes inputted by the user have been saved in two variables respectively named ‘node1’ and ‘node2’, without any casting of the variables or any unpacking the variables can be inputted into the distance function where the .miles appendix is added in order to specify the measurement unit and then the distance can be calculated and outputted.
In order to keep some accuracy the distance was rounded to two decimal places.

**Prototype for the statistics tab:**

```python
from geopy import distance
import customtkinter as ctk

ctk.set_appearance_mode('dark')
ctk.set_default_color_theme('blue')

class calculation(ctk.CTk):
    def __init__(self):
        super().__init__()

        self.geometry('800x600')
        self.title('node calculations')

        self.grid_columnconfigure(1, weight=1)
        self.grid_rowconfigure(1, weight=1)

        self.calcFrame = ctk.CTkFrame(self, corner_radius=10)
        self.calcFrame.grid(row=0, row=0, columspan=2, rowspan=2, padx=10, pady=10, sticky='nsew')
        self.calcFrame.grid_columnconfigure((0,1), weight=1)
        self.calcFrame.grid_rowconfigure((0,1), weight=1)
        self.calcFrame.propagate(False)

        self.nodeEntry1 = ctk.CTkEntry(self.calcFrame, placeholder_text='input node1 here')
        self.nodeEntry1.grid(row=0, column=0, padx=10, pady=10)

        self.nodeEntry2 = ctk.CTkEntry(self.calcFrame, placeholder_text='input node2 here')
        self.nodeEntry2.grid(row=0, column=1, padx=10, pady=10)

```

```
        self.calcDist = ctk.CTkButton(self.calcFrame, text='calculate distance', command = self.calculations)
        self.calcDist.grid(row=1, column=0, padx=5, pady=5)

        self.unitOpt = ctk.CTkComboBox(self.calcFrame, values=['miles', 'kilometres'])
        self.unitOpt.grid(row=1, column=1, padx=10, pady=10)

    def calculations(self):

        global node1 ; node1 = self.nodeEntry1.get()
        global node2 ; node2 = self.nodeEntry2.get()

        if self.unitOpt.get() == 'miles':
            d = distance.distance(node1, node2).miles
            print(d)
        elif self.unitOpt.get() == 'kilometres':
            d = distance.distance(node1, node2).kilometers
            print(d)

if __name__ == '__main__':
    app = calculation()
    app.mainloop()
```

Here is the updated object oriented prototype code for the statistics node distance calculations, later on fuel consumption and fuel cost will also be calculated and added to the code.

**Miscellaneous updates:**

In terms of the clear route button I've updated the subroutine to include lines that will also remove the markers and paths from the map overlay on the screen.

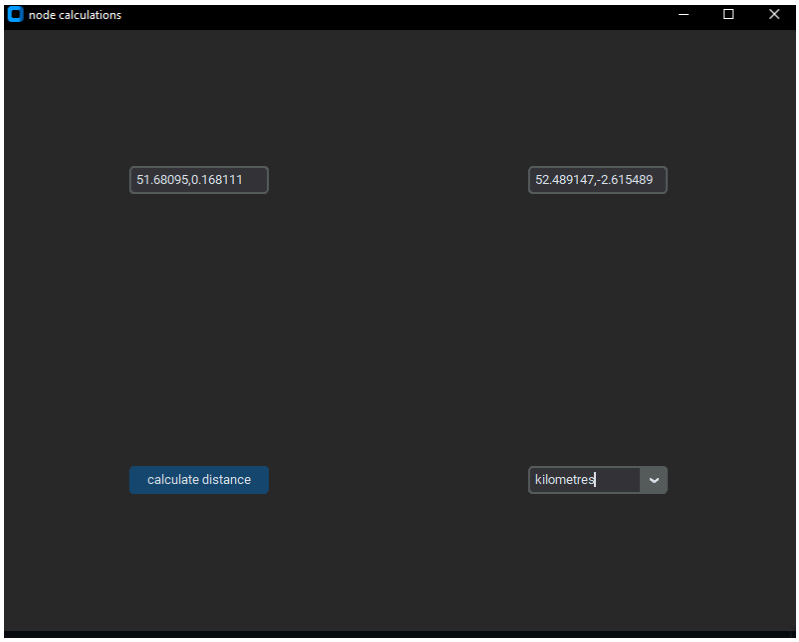
```
def clearInput(self):
    with open('arfrouteinfo.txt', 'r+') as dataFile:
        dataFile.truncate(0)
    self.overlay.delete_all_path()
    self.overlay.delete_all_marker()
```

The settings tab has been also updated, the option menu in order to select the data type for the node calculations has now been removed as the new geopy model will only be taking float/real values and integer values for the input so no configuration of the input type by the user is needed.
Furthermore the option menu for the unit measurements has been added to the settings tab where through the calculations procedure the user can define whether they want their route distance to be calculated in miles or kilometres.

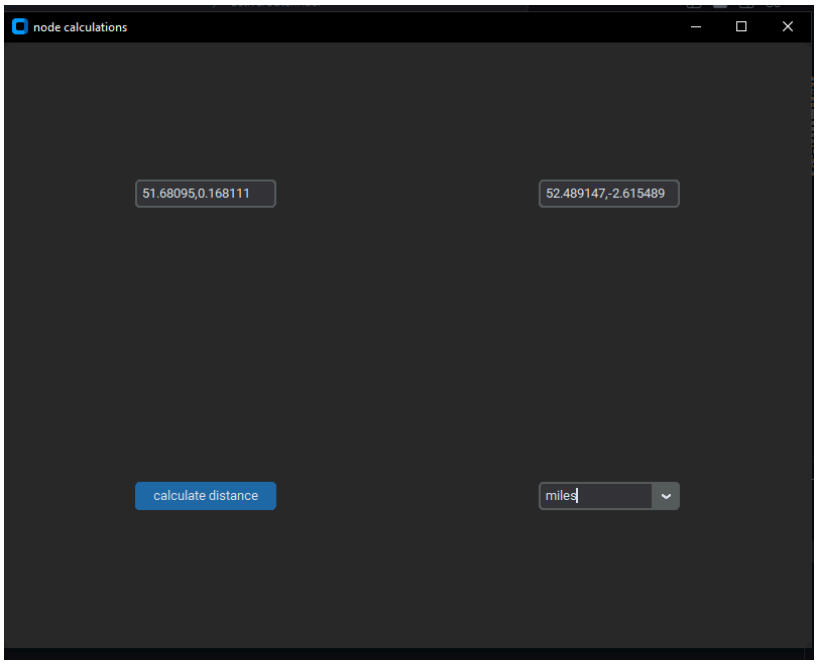
```
#self.valueInput = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values = ['float', 'string'])
#self.valueInput.grid(row=1, column=1, padx=5, pady=5)

self.unitLabel = ctk.CTkLabel(self.multiBox.tab('settings'), text='configure unit measurements')
self.unitLabel.grid(row=2, column=1, padx=10, pady=10)

self.unitOpt = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values=['miles', 'kilometres'])
self.unitOpt.grid(row=3, column=1, padx=10, pady=10)
```

**Here is the output:**

The first one has been calculated in kilometres according to the combo box being selected on that option and following it is the output for the distance between the two nodes inputted.
The second test has been calculated in miles according to the combo box being selected on that option and following it is the output for the distance between the two nodes being inputted.








The distances have been calculated and outputted onto the command interface, however on the main software iteration the result will be rounded and displayed on the user interface as a label as well as a appendix stating whether the calculated distance is in miles, kilometres or even nautical miles, due to the geopy module calculating the distance based off of the spherical model of the earth, the error margin for the calculated distance will be <0.5% which is still small enough in order to produce accurate results for the users and subsequent calculation for fuel consumption and costs, etc.

For implementation of the object oriented script into the active route finder some adjustments had been made to the configuration of the menu layout and functionality.

## Stage 1 (User Interface Creation) Review:

The first stage consisted of creating a base for the program via a graphical user interface, this user interface is a major part of the program, allowing users to interact with the program directly without the use of a command line interface which would have been tedious and time consuming.
As outlined in the design section the user interface was created with usability in mind so that navigating the program would be no issue for the user,  a major parent class was used in order to group together user interface widgets.
So far on this iteration of the graphical user interface a successful tabview section has been created on one half of the window where users will be able to input data and view information about the route, while the other half contains the map overlay, whereby users will be able to directly input nodes onto the map

How it was tested:

The user interface sections were created on a separate file where they were isolated to the main program, therefore functionality could be tweaked with and customised while not directly affecting the progress and widgets of the active route finder graphical user interface.
As widgets such as the frame were created they were placed onto the corresponding grid columns and rows of the windows main frame that contains all of the other smaller frames, which in turn contain all of the widgets for user interactions.

How it meets success criteria and user expectations:

As mentioned in the design section I would ensure that the user interface would be accessible and easy to use, to ensure this I kept the graphical user interface simple using the customtkinter module to

add accessibility options such as light mode or dark mode in order to aid the user experience, as well as this adding multiple user interface scaling values allows the user to configure the size and proportions of the active route finder to their desired settings.
As well as the promised active route finder user manual which provides details to the user on how to run and use the program should they have any issues, this improves accessibility and usability of the program by providing offline static support that can be accessed by the user any time.

Specific criteria being met:

- Accurate and up to date map
- Easy to navigate user interface
- Easy to understand user interface
- User manual

Changes from the design of stage 1 (graphical user interface):

For the nodes on the graphical user interface if you were to hover or click them information about the node was promised to be shown, however as development progressed I realised that this function was more a part of stage 2 than stage 1 and therefore was not included within the stage 1 scripts.
As well as this the providers for the map overlay in the user interface was initially to be a OpenStreetMap API that the users could interact with, however as the development progressed a more efficient module (TkinterMapView) provided more functionality within the customtkinter module and integration with the graphical user interface than OpenStreetMap could have done.
Summary of the project as a prototype iteration at this stage:

So far the user interface has been successfully created with the user accessibility, all of the user interface widgets have been placed and created using the .grid() methods, furthermore the tkintermapview module was successfully implemented and integrated with the customtkinter user interface allowing an interactable map to be accessible to the user within the program, as well as this user interface tabs for the route log, data section, user manual and settings were also created successfully.

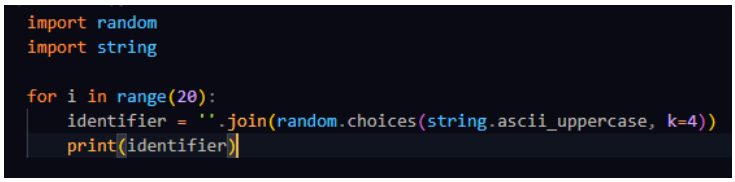
**Further testing and validations:**

The user interface has many different components that overlap and communicate with each other, therefore the functionality and compatibility of these separate components must be tested to ensure the final product (Active Route Finder) works flawlessly.

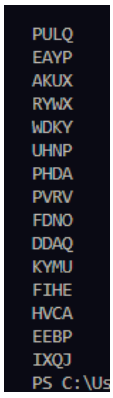
**Stage 2 (Program Functionality):**

The second stage of the active route finder will consist of iterative development for the functionality that will bind to the graphical user interface widgets in order to create one fully functioning program.

**Prototype code for the unique identifiers:**



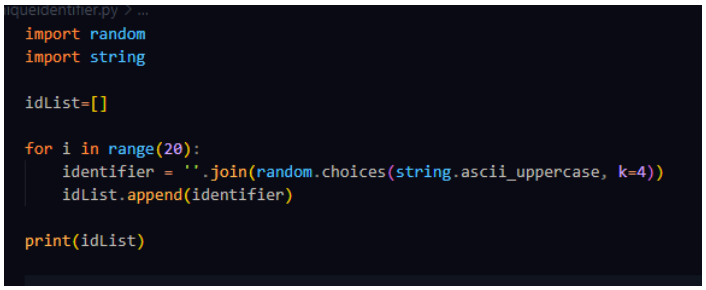

The modules random and string have been imported into the script in order to concatenate random ascii characters together making an unique identifier which can be linked to nodes for easier identification, the for loop is used with a range function in order to limit the amount of unique identifiers that are created, later on the range will match the number of nodes the user has inputted into the program just before the start button has been pressed.




**Test printing the unique identifiers:**

As you can see here due to the k=4 command the unique identifier characters have been restricted to 4 only, after these unique identifiers have been created the idea is to store them in a list, array or stack of some sort after which they can be assigned one by one to a node.
Now a list of unique identifiers have been successfully created, from here we will work on binding them with nodes and displaying them onto the map overlay as well as route logs and external text files.

**Update 1:**




Here an empty list has been created where after the identifier is appended to the list and due to the for loop the list becomes populated with 20 items

**Test print:**


As a result printing the list after the for loop will give this result, from here we can access individual values and then assign them to a node.

However on the node side we may have to tweak some of the code in order to make sure that the assignments run smoothly.

**Update 2:**

```
import random
import string

idList=[]
nodeList = ['node1', 'node2', 'node3', 'node4', 'node5', 'node6', 'node7', 'node8', 'node9', 'node10']

for i in range(10):
    identifier = ''.join(random.choices(string.ascii_uppercase, k=4))
    idList.append(identifier)

print(idList)



y = 0
while y < 10:
    print(nodeList[y] +' , ' +idList[y])
    y = y+1
```

Here is another iteration of the string identifier code. I created a list with some placeholder nodes where they are named incrementally, after the previous code a while loop was created in order to cycle through the lists.
Initially I thought of creating two separate while loops, one for the nodeList and one for the idList, however to improve the efficiency of the program I settled with creating one while loop in order to cycle through both of the lists simultaneously.
From there on string concatenation is used in order to conjoin the cycled values of the two lists, using y=y+1 allows the list to keep on cycling through the while loop until it reaches a preset number of loops which here would be 10, in the final iteration the number of loops should correspond with the amount of nodes that the user has placed down into the software.

**Test Output:**

```
['KGOR', 'RWII', 'NOIF', 'XFAU', 'BPAV', 'QCBF', 'WVTG', 'RBZM', 'YEWE', 'NVQS']
node1 , KGOR
node2 , RWII
node3 , NOIF
node4 , XFAU
node5 , BPAV
node6 , QCBF
node7 , WVTG
node8 , RBZM
node9 , YEWE
node10 , NVQS
```

And now the nodes, alongside their own unique identifier have been outputted.
In order to adapt this for the final iteration of the active route finder, the concatenated lists must first of all be outputted to the route log as well as inputted into the tkinter map view position function in order to provide the location of the node as well as the text which in this case will be the unique identifier.
Furthermore I need to make sure that the nodeList is updated with the coordinates that the user inputs as a tuple into the data section so the nodeList will have to contain tuples, instead of singular items. From here the nodeList can be appended with the tuple of every coordinate that the user inputs into the active route finder.

**Updated node input system:**

Tkintermapview has a function that allows events to be binded to the left and right mouse buttons when they are clicked. In order to improve the usability of the software as well as improving the efficiency of the program.

```
def leftClick(self, coordinates_tuple):

    global nodeList ; nodeList = []
    idList = []
    global currentNode ; currentNode = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
    #rounded = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
    nodeList.append(currentNode)



    for i in range(1):
        identifier = ''.join(random.choices(string.ascii_uppercase, k=4))
        idList.append(identifier)

    x = 0
    while x < 1:
        NodesId = []
        totalList = nodeList[x], idList[x]
        print(totalList)
        x = x + 1



    self.overlay.set_marker(coordinates_tuple[0], coordinates_tuple[1], text=identifier)
```

Joining together the unique identifier creator and lists into one function allows us to create a multipurpose function that takes the input of coordinates from the left click mouse event on the map overlay.
From here the coordinates are rounded and then appended to a list which then after the for loop for the unique identifier is initiated, for the range unique identifiers are created which are then appended to the nodes in the nodeList. Therefore creating a tuple of the node coordinates and its very own unique identifier.
Finally the last line of this script then takes the coordinates of the area on the map where the left click event occurred and then places a node onto the coordinates selected.

This is more efficient than asking the users to input nodes manually and can decrease the error rate of user inputs and node placements.

```
self.overlay.add_left_click_map_command(self.leftClick)
```

Here is the command that is referenced by the function in the class body.
It is connected to the tkintermapview overlay and therefore the command is able to place a marker for the node directly to the map overlay.

Previously the list for the nodes, identifiers and both of them combined were stored as list=[] within the procedure itself, which caused the list to reset everytime the procedure was called, as a result when the currentNode variable was appended to the list only the most latest addition was shown and not all of the currentNode additions to the lists.

```
def leftClick(self, coordinates_tuple):
    global nodeList ; nodeList = []
    global idList ; idList = []
    global currentNode ; currentNode = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
    nodeList.append(currentNode)
```

As you can see the global list variables are being set within the procedure for the left click event.

```
['DBPT']
((52.5169, 13.376), 'DBPT')
[(52.5169, 13.376)]
[(52.5169, 13.376)]
['XYXO']
((52.5183, 13.3763), 'XYXO')
[(52.5183, 13.3763)]
[(52.5183, 13.3763)]
user has exited the active route finder
```

**Test print:**
Here is the output on the command line made for testing the list behaviour, as you can see everytime a new node was inputted through the left click event the node and identifier list are resetted.

In order to fix this list reset issue I instead took the lists and added a global scope to them and placed the list variable within the parent class itself, therefore every time the program is started the list is empty and then it can be appended to through the procedure:

```
    #creating lists with the global scope in order to be used in the route procedures

    global pathList ; pathList = []
    global nodeList ; nodeList = []
    global idList ; idList = []
    global positionList ; positionList = []

#creating functionality for the change theme button

def changemode(self, newmode: str):
    ctk.set_appearance_mode(newmode)
```

Here's the updated output where the list is now being correctly appended to:

```
((52.5189, 13.3761), 'YOKR')
[(52.5189, 13.3761)]
((52.5189, 13.3761), 'YOKR')
[(52.519, 13.377), (52.5189, 13.3761)]
((52.5189, 13.3761), 'YOKR')
[(52.5189, 13.3778), (52.519, 13.377), (52.5189, 13.3761)]
user has exited the active route finder
```

**Test print:**
As you can see every time a node is selected through the left click procedure, the coordinates are now being correctly appended to the list and outputted in its entirety.

However one issue still remains, this is the second to last list on the command line is the totalList which can be seen within the leftClick procedure. Using a while loop with the range set to one, the nodeList containing the rounded node data and the idList containing the randomly generated unique identifiers are concatenated together and then output for testing purposes.
However as can be seen in the image above the totalList only outputs the currentNode and its identifier which was the same issue with the positionList from the last error.

**Here is procedure section containing the totalList:**

```
x = 0
while x < 1:
    global totalList ; totalList = nodeList[x], idList[x]
    x = x + 1
print(totalList)
```

The issue pertaining to the list printing the most recent index appended to the positionList could be due to the while loop. Before the loop is initiated, the variable x is set to zero, from here the while loop takes in the command of creating a global scoped list (totalList) then setting the list equal to the corresponding x value index of the separate nodeList and idList. After this the x value is incremented in order to ensure that in the next while loop when the next left click event occurs that the next indexes are selected, which will be the most recently appended.
In order to solve this problem we will have to look at a similar solution as the issue with the positionList, in that instance the global scoped list variable was placed within the procedure causing a list reset every time the procedure was called upon, the case is similar here.
Therefore I will take the global scoped totalList variable and instead insert it in the class body so it is empty when the program is started and is appended to every time that the procedure for the left click event is called to.

```
x = 0
while x < 1:
    totalList.append(nodeList[x], idList[x])
    x = x + 1
print(totalList)
```

```
#creating lists with the global scope in order

global pathList ; pathList = []
global nodeList ; nodeList = []
global idList ; idList = []
global positionList ; positionList = []
global totalList ; totalList = []
```

In the while loop instead of creating the list with the nodeList and idList indexes, it now simply appends those indexes to the already created list.

```
  File "C:\Users\hamza\ActiveRouteFinder\GUIV2.py", line 243, in leftClick
    totalList.append(nodeList[x], idList[x])
TypeError: list.append() takes exactly one argument (2 given)
```

However an issue arises when the left click event is activated on the map overlay, the issue reads that a tuple (in this case the nodeList and idList indexes) cannot be appended to the totalList as the .append() method only takes one argument and in this case more than one has been given.
In order to solve this we'll have to find a way to either allow list appendages to accept tuples or either to turn the totalList tuple into a string or one data item.


```
x = 0
while x < 1:
    totalList.append(tuple[nodeList[x], idList[x]])
    x = x + 1
print(totalList)
```

In order to solve the issue I labelled the append input for the tuple with the tuple keyword which contained the tuple I wanted to input into the totalList, the rest of the procedure section has been kept the same.
The nodes on the left represent the test nodes I inputted in order to test the outcome of the list appending.

```
[tuple[(52.5168, 13.3771), 'ZZOA']]
[(52.5168, 13.3771)]
[tuple[(52.5168, 13.3771), 'ZZOA'], tuple[(52.5168, 13.3771), 'ZZOA']]
[(52.5171, 13.3786), (52.5168, 13.3771)]
[tuple[(52.5168, 13.3771), 'ZZOA'], tuple[(52.5168, 13.3771), 'ZZOA'], tuple[(52.5168, 13.3771), 'ZZOA']]
[(52.5155, 13.3768), (52.5171, 13.3786), (52.5168, 13.3771)]
```

Here we can see the positionList (list that contains the unique identifier) is being appended to as expected, however the issue is that the items being appended are simply duplicated over and over from the initial first item within the positionList, therefore this issue may be related to the list appending commands contained within the while loops or the left click event retrieving the current nodes from the map overlay.
In order to solve this we are going to have to analyse and dissect the working of the while/for loops and the coordinate tuple retrieved from the map overlay.
First of all a global variable is declared, named currentNode. This variable contains the coordinate tuple indexes for the first and second position coordinates corresponding to x and y retrieved from the map overlay whenever the left click event is activated.
After this the variable currentNode is appended to the empty nodeList and then populates it every time the left click event is activated. After this the list is printed to show this result.

**Fixing the appendix duplication issue:**

```
totalList.append(tuple[nodeList[-1], idList[-1]])
print(totalList)

fileCommand = open('arfrouteinfo.txt', 'a')
fileCommand.write(str(currentNode))
fileCommand.write(identifier)
fileCommand.write('\n')
fileCommand.close()
```

This is the fix that was created, instead of as before creating a for loop with an x value that increments to give the new index of the item in the list, instead we now append the last items of the referenced node and id lists, this cuts down the lines for the same function.
To ensure the integrity between the totalList and what is appended to the arfrouteinfo text file I included the file commands that append the node info to the text file.

```
[(52.517, 13.3731), (52.5149, 13.3743), (52.5169, 13.3752)]
[tuple[(52.517, 13.3731), 'QTCG'], tuple[(52.5149, 13.3743), 'KBSW'], tuple[(52.5169, 13.3752), 'EQGZ']]
```

Now we can successfully see the command line output producing the correctly appended totalList in tuple form with coordinates and unique identifier on each tuple.

```
(52.517, 13.3731)QTCG
(52.5149, 13.3743)KBSW
(52.5169, 13.3752)EQGZ
```

Here the arfrouteinfo text file has also been correctly edited with the coordinates and the unique identifier of the nodes that have been added through the left click event.

```
  f = open('arfrouteinfo.txt', 'a')
  f.write(nodeList)
  f.write('\n')
  f.close()

ef clearInput(self):
  with open('arfrouteinfo', 'r+') as dataFile:
      dataFile.truncate(0)
  self.overlay.delete_all_marker
  self.overlay.delete_all_path

ef calcDist(self, coordinates_tuple):
  if self.unitOpt == 'miles':
      dist = distance.distance(currentNode, currentNode).miles
  elif self.unitOpt == 'kilometres':
      dist = distance.distance(currentNode, currentNode).kilometers
  print(round(dist, 4))
```

The calculation for the distances is still in work, but in prototype it should receive the coordinates tuple from the left click event and then compare two nodes in the nodeList which have been subsequently appended in order to input into the geopy distance function and returned will be the distance either in kilometres or miles of the distance between the two nodes.

Here's the test code created to solve the problem:

```
from random import randint

distList = []
totalDist = 0

for i in range(5):
    value = randint(1,100)
    distList.append(value)
    if len(distList) > 1:
        totalDist = totalDist + distList[-1]
        print(totalDist)

print(distList)
```

To make a populated list I simply added random integers to the end of an empty list, from here in the for loop after every loop was completed the if statement would check whether the lists length is longer than 1 (this is to ensure that a total can be retrieved from 2 values), after this the totalDist variable which is initially set to 0 is now added with the last value of the distList, from here the running total of the totalDist is printed at the end of every loop, this loop only runs 5 times, however the loop in the active route finder would run as many times there are nodes placed.

```
84
139
230
318
[14, 84, 55, 91, 88]
PS C:\Users\hamza\ActiveRouteFinder>
```

This is the command line output, where the running total is outputted 5 consecutive times, as we add the subsequent values by hand we can see that it corresponds with the running total values that have

been outputted and therefore a running total for the total distance of the journey has now been calculated.

In order to adapt this prototype for the active route finder a couple variables and statements will have to be changed, such as the variable names, but furthermore there will have to be a selection aspect as the program allows the user to select whether they want the distance to be in miles or kilometres. As a result the initial for loop is going to be branched into two sections, one for miles and one for kilometres.

```
def leftClick(self, coordinates_tuple):

    global currentNode ; currentNode = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
    nodeList.append(currentNode)

    for i in range(1):
        global identifier ; identifier = ''.join(random.choices(string.ascii_uppercase, k=4))
        idList.append(identifier)

    totalList.append(tuple[nodeList[-1], idList[-1]])

    #create marker from the selected coordinates from the left click event on map overlay
    self.overlay.set_marker(coordinates_tuple[0], coordinates_tuple[1], text=identifier)

    #simulateanously open text file and write coordinate data and unique identifiers to that text file
    fileCommand = open('arfrouteinfo.txt', 'a')
    fileCommand.write(str(currentNode))
    fileCommand.write(identifier)
    fileCommand.write('\n')
    fileCommand.close()

    fileRead = open('arfrouteinfo.txt', 'r')

    #read the coordinate and identifier data from arfrouteinfo.txt in order to display to the route log
    self.logText.insert('0.0', fileRead )

    fileRead.close()

    #integrate a running total for distance in left click event
```

```
if len(nodeList) > 1:
    for x in range(1):
        if self.unitOpt == 'kilometers':
            dist = distance.distance(nodeList[-1], nodeList[-2]).kilometers
            distList.append(dist)
            if len(distList) > 0:
                totalDistance = sum(distList)
                print(nodeList)
                print(idList)
                print(totalList)
                print(distList)
                print(totalDistance)


        else:
            dist = distance.distance(nodeList[-1], nodeList[-2]).miles
            distList.append(dist)
            if len(distList) > 0:
                totalDistance = sum(distList)
                print(nodeList)
                print(idList)
                print(totalList)
                print(distList)
                print(totalDistance)
```

This is the complete version of the distance sums that have been adapted and added to the active route finder

Here is a test of nodes that were imputed into the active route finder:

```
[tuple[(52.5172, 13.3768), 'YZYP'], tuple[(52.5146, 13.3768), 'XZAX'], tuple[(52.5109, 13.3768), 'MHOV']]
[0.17977545364204026, 0.2558341624296937]
0.435609616071734
```

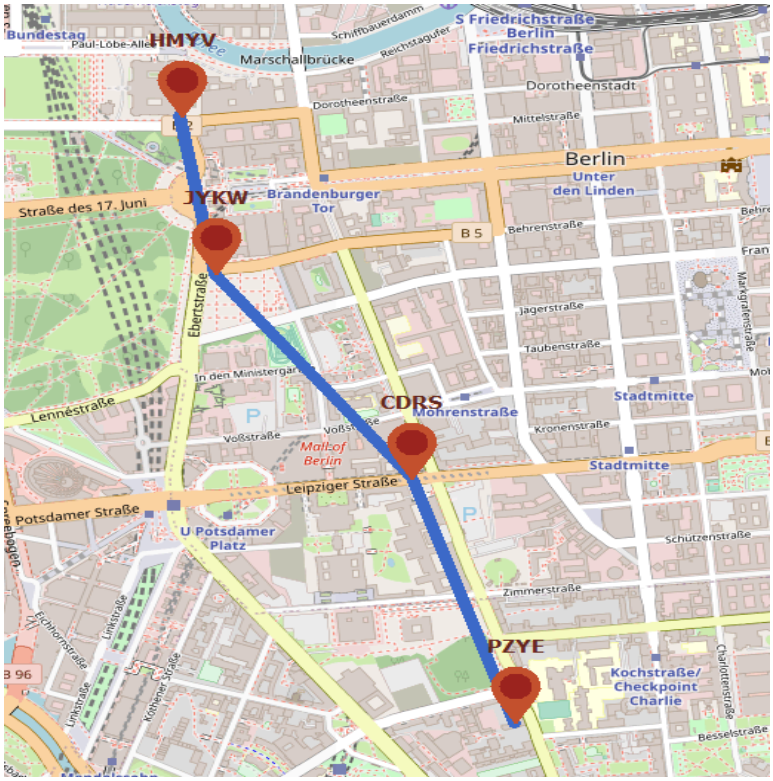
The first list is the tuple totalList that has been created with the 3 nodes added, with their corresponding coordinates and unique identifiers, below this list is the distList containing the distances from one node to another, by using the sum function for the distList we were able to produce the final value which is the total distance across all nodes in miles in this circumstance.

**Create Path animation for user:**

In order to allow the user to see the flow of their route and how to follow their route plan I included tkintermapview's set_path function that allows positions or markers on the map overlay to be connected with a path, the path list is derived from the nodeList and the path is connected through these coordinates.

```
        if len(nodeList) > 1:
            self.overlay.set_path(nodeList)
```

Here's  a test of the path connections and as you can see the path connects each node placed.

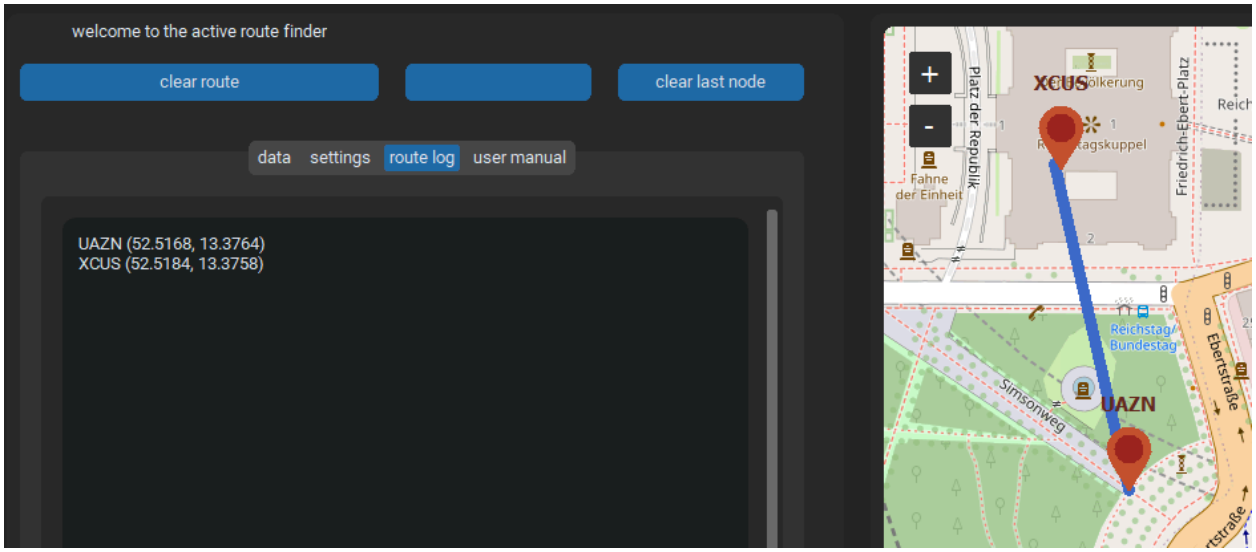



**Route Log Functionality:**

Here I have created some insert code for the logText which is a textbox on the route log that displays the collection of nodes and coordinates to the user.
Using some simple '\n' commands and list indexes I was able to turn the indexes into strings that could then be inserted into the position at line 0, character 0.

```
#logText inserts for coordinates and identifiers

self.logText.insert('0.0', '\n')
self.logText.insert('0.0',  str(nodeList[-1]))
self.logText.insert('0.0', ' ')
self.logText.insert('0.0', str(idList[-1]))
```

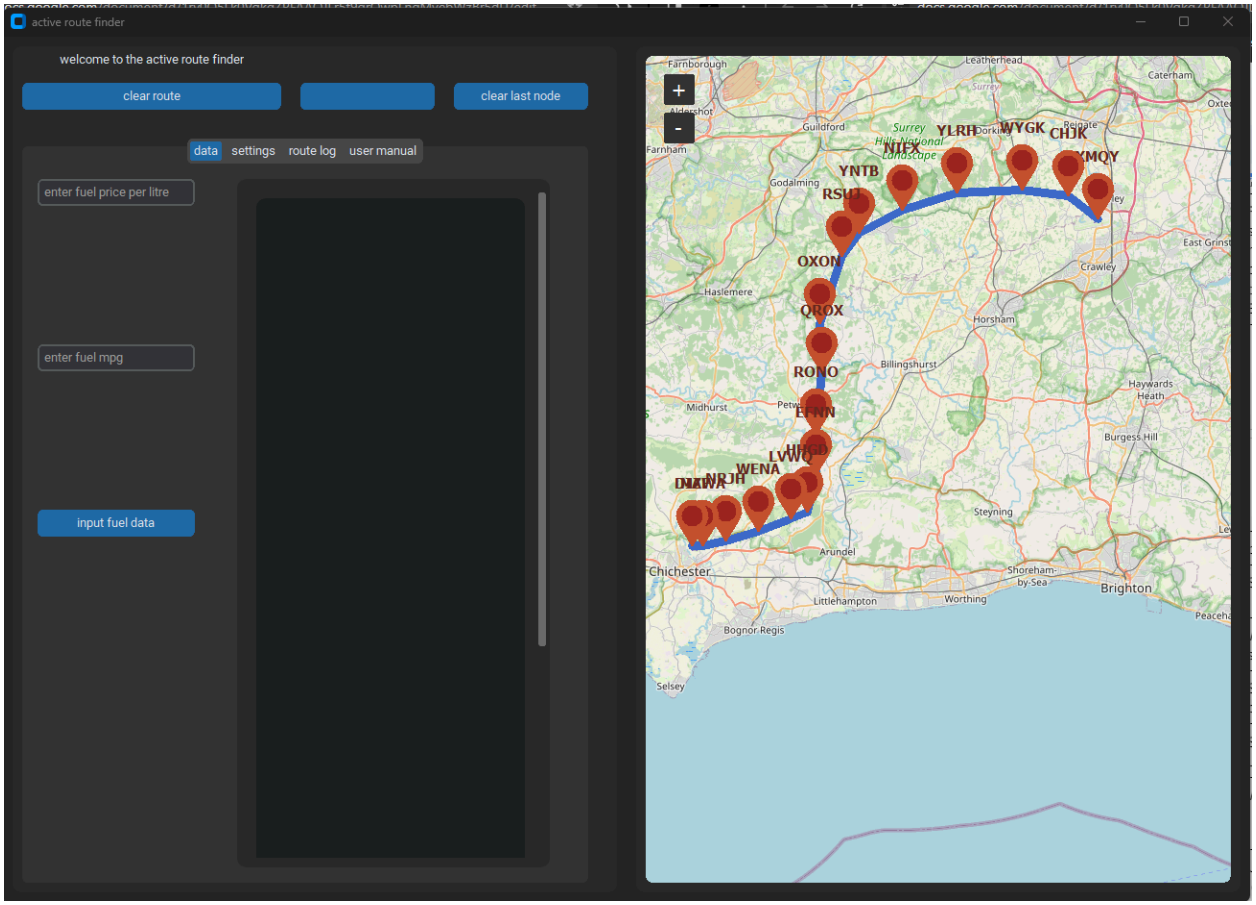
**Testing Functionality:**
The user interface is shown here, some nodes were placed down onto the map overlay and the corresponding values appeared onto the route log, this feature will allow the user to keep track of their nodes in long journeys and follow the route through each individual node.



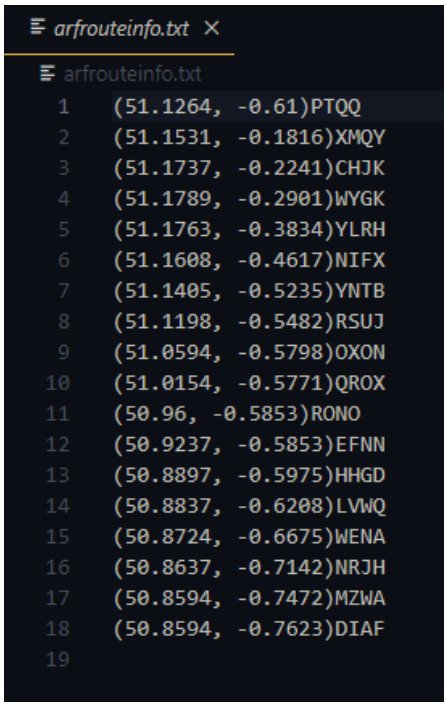

Here is the total list, coordinate list and identifier list which also show the corresponding values on the route log which indicates that the correct values have been added.

```
[(52.5184, 13.3758), (52.5168, 13.3764)]
['XCUS', 'UAZN']
[tuple[(52.5184, 13.3758), 'XCUS'], tuple[(52.5168, 13.3764), 'UAZN']]
[0.11348898333326142]
```

**Testing the clear node functionality:**

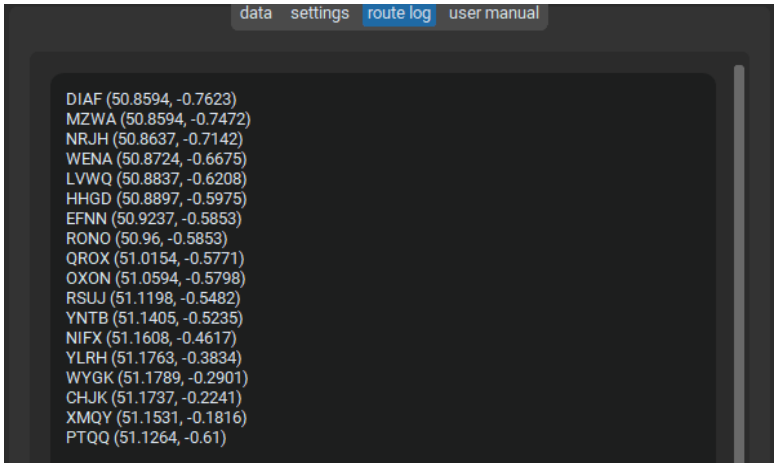



Here I have created a test route with waypoints between two airports in the south of England.



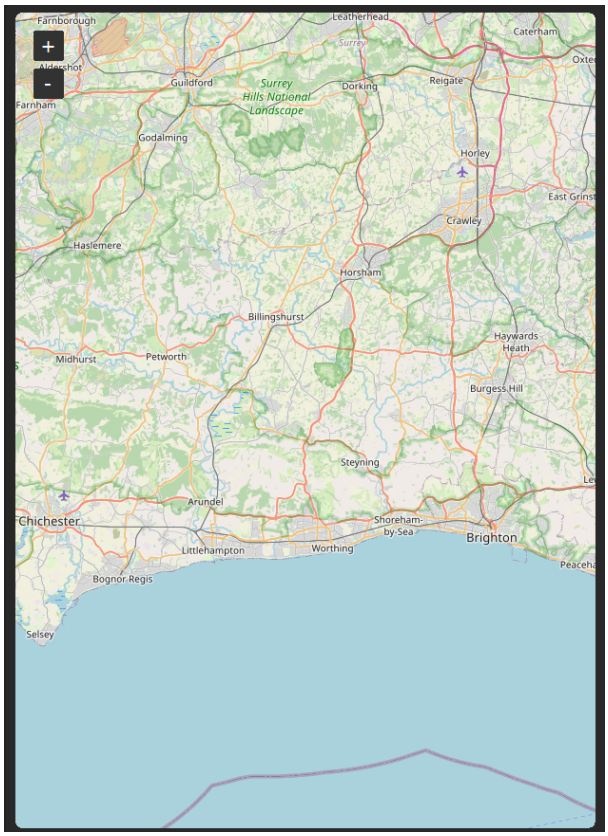

Here is the text file output.

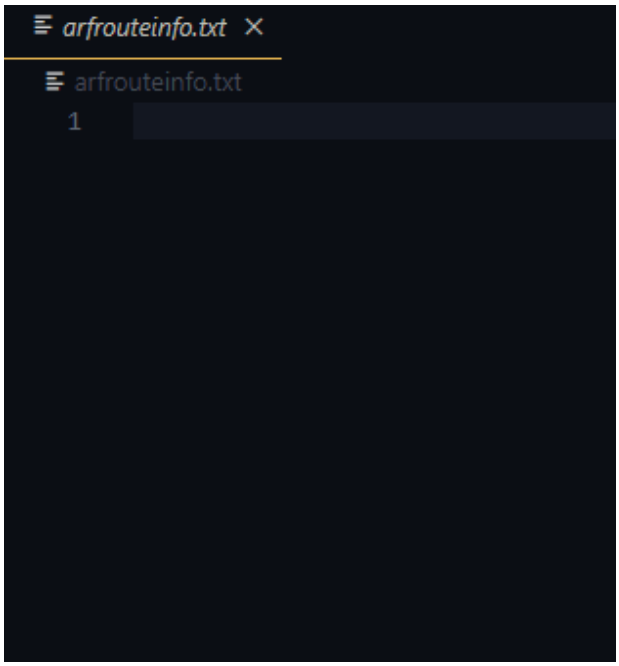
The data from the route identifiers and coordinates have been correctly plotted and transferred to the external text file.



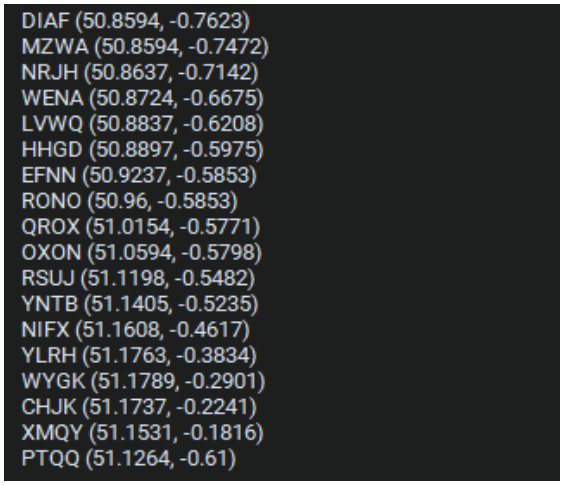
The data from the external text file was also correctly transferred to the route log.

Now I will activate the clear route button and record the results




The nodes from the map overlay and the unique identifier and coordinate data from the external text file have been removed successfully.




However the route log still retained the waypoint and route information which could confuse users when they go to create a new route as a result this test was partially passed in aspects of the map overlay and external text file, however some maintenance could be used on the route log to ensure that the data is correctly removed in order to improve usability.

## Data Validations:

In order to ensure that the data being entered into the software is being validated and is at the right type, it must be validated.
I will be using procedures and the try/except error handling  method in order to ensure that areas where data will be inputted such as at the fuel price input.

```
priceIn = input('input fuel price per litre: ')

if priceIn.isnumeric() == 'True':
    print('valid input')
else:
    print('invalid input')
```

Here is an early iteration of the data validation process for the fuel input so far using the .isnumeric() method we are checking whether the string inputted contains numbers, however one issue with this so far is that it will not take into account any decimal points that are inputted into the string, therefore we will have to create an exception for this.
When this code is ran however using random positive integers this is output:

```
input fuel price per litre: 100
invalid input
```

The error I identified was that the condition for the numeric function to equal to True was in quotations which caused it to act as a string, this can be easily be fixed by removing the quotations.

```
input fuel price per litre: 10
valid input
```

Now to improve the accuracy of the validation we need to limit the length of input to the average which is around 6 characters long including the decimal point which in this case will be mandatory. Therefore we can create a format check for the input specifying a specific way for the price to be input and making sure that the user follows that way.

**Fuel consumption and costs calculations:**

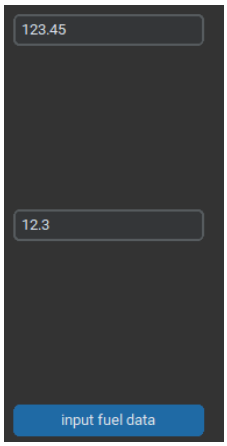
In order to calculate the fuel consumption and costs we are going to first have to pull the value for the average daily fuel price for the UK.
This can be done by asking the user to manually input the average fuel price for the country into the software, another option could be to web scrape the data, however the complexity would exceed the benefits of it.

To calculate fuel consumption we divide the distance travelled by the fuel efficiency measured in miles per gallon, after this to calculate the fuel cost we multiply the fuel consumption by the fuel price per litre.

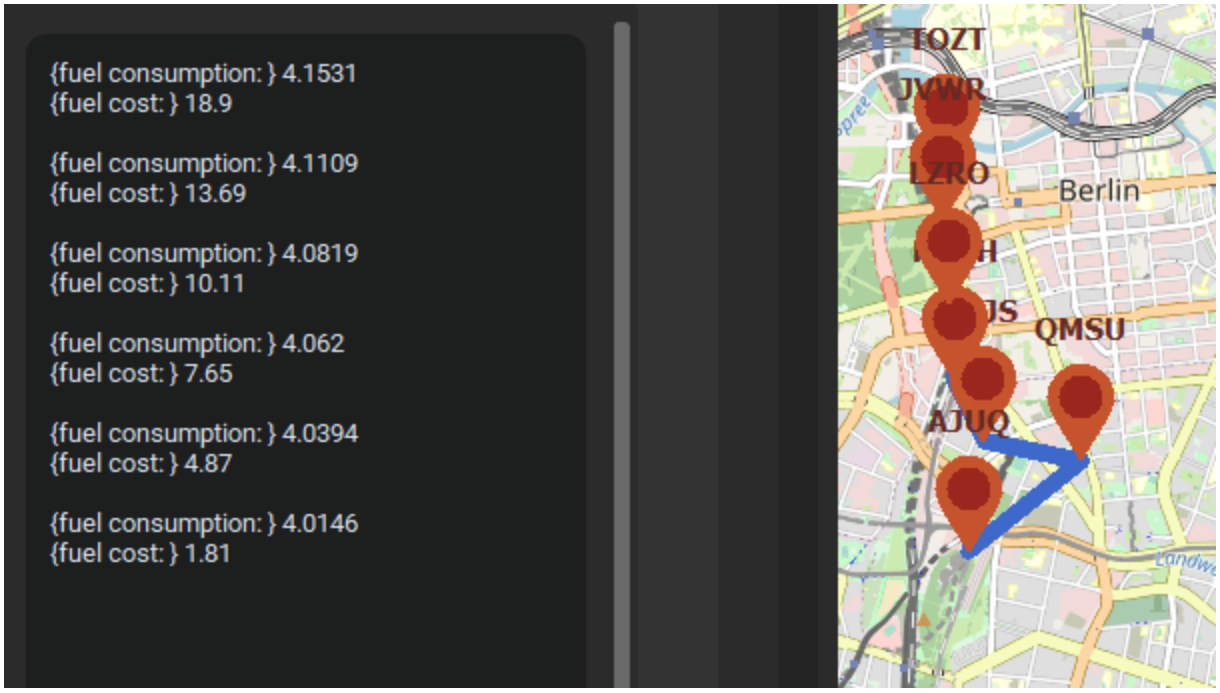
This block of code below is embedded within the left click event subroutine and is written twice in the branches for the miles and kilometres measurement units.
It works firstly by using the .get() method to gain the user inputs from the input boxes on the data tab, then using the predefined calculations, the fuel consumption and cost are calculated and then appended to a predefined global list, they are rounded to keep simplicity then the sum of the lists are calculated and rounded before being created into a string variable which is then finally added to the text box on the data tab through the .insert() method defined by tkintermapview.
Some issues along creating this code included different variable castings which interfered with the calculations as well as the initial total not being outputted, therefore lists had to be made to calculate the sums and output them to the textbox.
Furthermore issues such as rounding methods not being able to be set when defining tuples also occurred.

```
if self.consumptionEntry.get() != '' and self.fuelEntry.get() != '':
    fuelMpg = self.consumptionEntry.get()
    # float(fuelMpg)
    # float(dist)

#   if self.fuelEntry.get() != '':
    fuelPrice = self.fuelEntry.get()
    # float(fuelPrice)

    fuelConn = float(dist) / float(fuelMpg)
    fuelCost = float(fuelPrice) * float(fuelConn)

    round(fuelConn, 4)
    round(fuelCost, 2)

    fuelConnList.append(round(fuelConn,4))
    fuelCostList.append(round(fuelCost, 2))

    sumFuelCost = sum(fuelCostList)
    sumFuelConn = sum(fuelConnList, 4)

    roundSumCost = round(sumFuelCost, 2)
    roundSumConn = round(sumFuelConn, 4)

    costOutput = str('fuel cost: '), roundSumCost
    fuelConnOutput = str('fuel consumption: '), roundSumConn

    self.dataTextbox.insert('0.0', '\n')
    self.dataTextbox.insert('0.0', '\n')
    self.dataTextbox.insert('0.0', costOutput)
    self.dataTextbox.insert('0.0', '\n')
    self.dataTextbox.insert('0.0', fuelConnOutput)
```

**Testing Functionality:**




Here we define some random test variables for the input fields the random test value for the fuel price in pence is 123.45.
While the random test variable for the fuel efficiency of the vehicle is 12.3 measure in miles per gallon.




After entering every node the consumption and cost values always increase, therefore inferring that the running total for these values is working.
Every new section of the consumption and cost values refers to the updated values after a new node has been added to the overlay, therefore the user can simply add in the price and efficiency values at the beginning of use and track their cost and consumption throughout the journey.

**Validations for input variables:**

As the data for the fuel consumption rate for the vehicle as well as the fuel price by litre is going to be inputted by the consumer, we are going to have to ensure that the data that is entered is formatted in a specific way and validated to ensure its accuracy or otherwise misrepresented route data about fuel cost, etc will be given to the user, or could cause the program to crash during one of the data calculations.
In order to keep the program from crashing I am going to be using try and except statements, these work by first stating try and then in the indented region placing the code we want to run (such as the validation and format checking code for the inputs) then after the indented block on an unindented line an except statement is created and then indented from that except statement is the code that will run if some issue occurs with the initial formatting and validation code.
Try statements are used on volatile code and allow them to be encapsulated so that an except statement can simply print out strings to the user instead of causing the program to crash and return

an error, we can use the except statement to notify the user that invalid data has been entered into the entry fields.

```
def validate(price):
    try:
        price = float(price)
        print('valid data input')
    except:
        print('invalid data inputted')

price = input('input fuel price per litre: ')

validate(price)
```

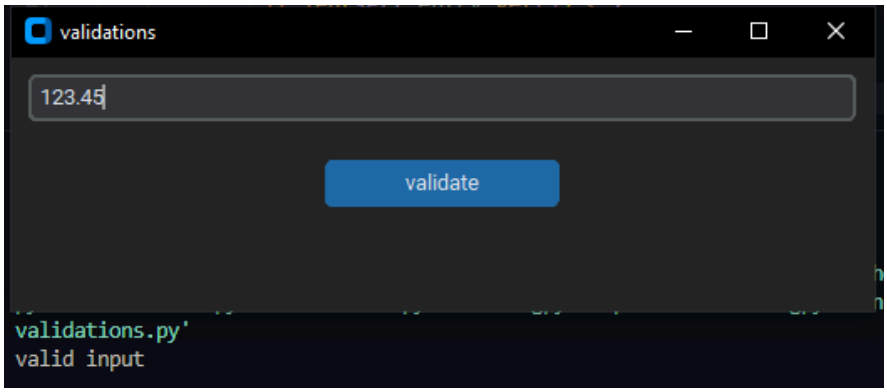
This is the initial iteration for the validation system for the fuel price input. A simple procedure has been created and by using the try and except statements to check if the input is numerical and float, from now we must create the validations for the length of the input which is intended to be limited to 6 characters including the decimal point.
To make sure that the validations are working correctly we are going to use some data inputs in forms of extreme and normal data.

Now we are going to convert the normal script into an object oriented script so that the results that are produced can be more accurate and any issues that may be related to an object orientation can be found and solved easily.

```
    def __init__(self):
        super().__init__()

        self.title('validations')
        self.geometry('800x600')

        self.grid_columnconfigure(1, weight=1)
        self.grid_rowconfigure(1, weight=1)

        self.entry = ctk.CTkEntry(self, placeholder_text='please enter input: ')
        self.entry.grid(row=1, column=1, padx=10, pady=10, sticky='new')

        self.button = ctk.CTkButton(self, text='validate', command = self.validations)
        self.button.grid(row=1, column=1, padx=10, pady=10)

    def validations(self):

        try:
            inputData = float(self.entry.get())
            print(len(str(inputData)))

            if len(str(inputData)) < 5:
                print('input too short')
            elif len(str(inputData)) > 7:
                print('input too long')

        except:
            print('invalid data input')

if __name__ == '__main__':
    app = display()
    app.mainloop()
```

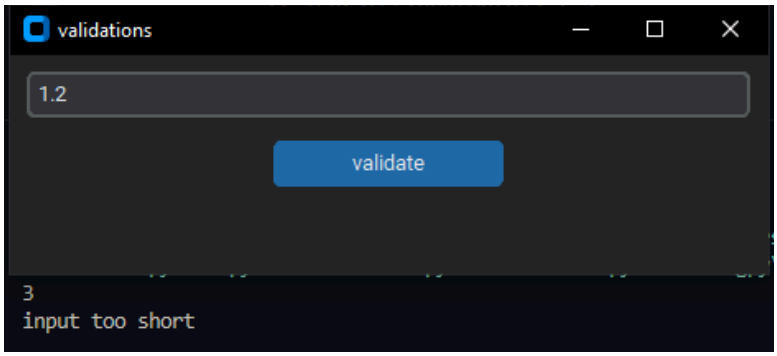
Here the object oriented script contains a procedure that takes the try/except statements from the last iteration of the validation script and then creates so that a command on a button can be inputted and therefore executes the validation procedure.
We ensure that the inputted data is of a certain length as well as a float value in order to cleanse and validate the data for the calculations later on.

**Test 1:**



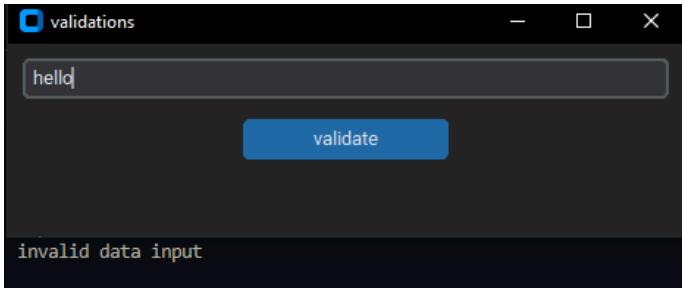

This first test used normal data, as well as boundary data that is in the data boundaries for the designated input.
The test went successfully and therefore the data was validated successfully.

**Test 2:**




Here the data used is extreme and outside the allowed parameters and as expected the test brings back a length warning to the user.

**Test 3:**




Here a non-numeric input has been submitted and the validations work successfully by alerting the user that the input is invalid.

Next we are going to formulate the equations and formulas for the fuel consumption and costs derived from the variables for the user input fields.

In order to calculate the fuel consumption first we must take the fuel consumption rate inputted by the user and then multiply it by the distance travelled through the route distance.
The route distance meanwhile has been calculated using the geopy module and some index manipulating to produce a float value for the distance between the nodes which can then be added to a running total and outputted entirely as the entire length of the route.

**Clearing Last Node Feature:**

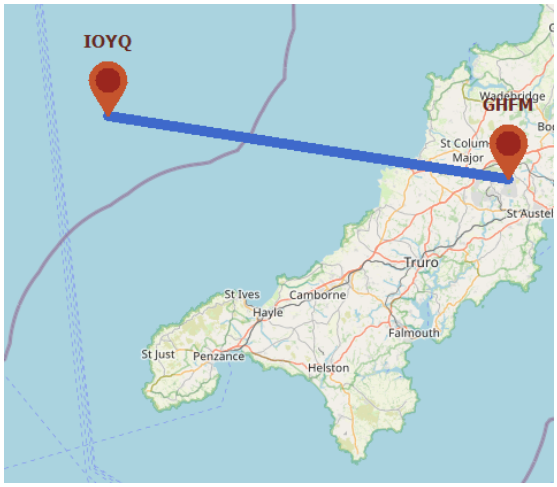
With this subroutine I intend to allow the user to delete their last placed node on the map overlay in case it was placed incorrectly or if they would like to change the position of the node.

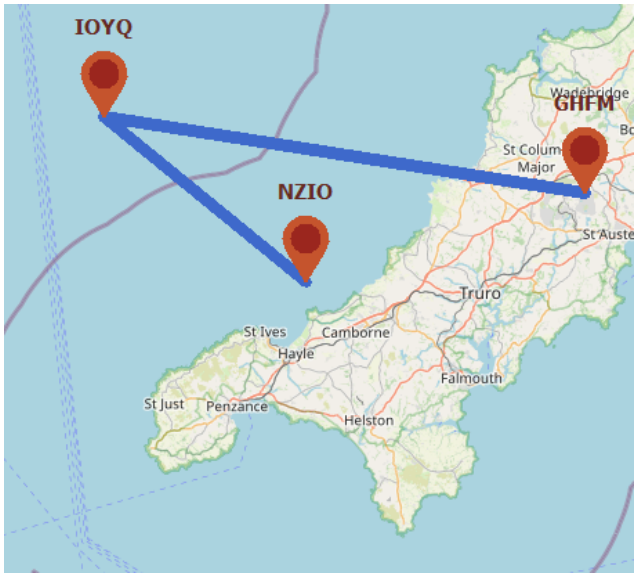
```
def clearLastNode(self):
  nodeList.pop()
  idList.pop()
  totalList.pop()
  marker.delete()
  path.delete()
```

Using the .pop() method that removes the last indexed item of a list, I was able to make it so that interacting with the clearLastNode button would remove the last items on the totalList, idList and nodeList.
And using the tkintermapview .delete() method I was able to make it so that the last placed marker and path would also be deleted from the map overlay.

**Testing Functionality:**

Here we can see the difference between the initial route placed and the second route after the clear last node button was pressed.
The first image shows the initial route.
The second image shows the route after the clear last node button was pressed.
The third image shows the route after a new node has been placed.


As coded the old node LMQX has been replaced with NZIO after another input, the path and marker that belonged between IOYQ and LMQX has been deleted and a new path and marker have been made between IOYQ and NZIO, NZIO being the new node that has been placed onto the map overlay.

```
[(50.3906, -4.8445), (50.4868, -5.8003)]
['GHFM', 'IOYQ']
[tuple[(50.3906, -4.8445), 'GHFM'], tuple[(50.4868, -5.8003), 'IOYQ']]
[42.712287385691766]
42.712287385691766
[(50.3906, -4.8445), (50.4868, -5.8003), (49.9084, -5.8799)]
['GHFM', 'IOYQ', 'LMQX']
[tuple[(50.3906, -4.8445), 'GHFM'], tuple[(50.4868, -5.8003), 'IOYQ'], tuple[(49.9084, -5.8799), 'LMQX']]
[42.712287385691766, 40.13289603275426]
82.84518341844603
[(50.3906, -4.8445), (50.4868, -5.8003), (50.2784, -5.3993)]
['GHFM', 'IOYQ', 'NZIO']
[tuple[(50.3906, -4.8445), 'GHFM'], tuple[(50.4868, -5.8003), 'IOYQ'], tuple[(50.2784, -5.3993), 'NZIO']]
```

Here is the command line which has the outputs of the lists, as expected when the clear last node button was pressed after the 10th line and a new node was inputted the coordinates and identifier for the node LMQX has been popped and removed from its respective lists, therefore this concludes that the functionality of the clear last node button is working correctly.

# Stage 2 (Functionality) Review:

In the second stage the main functionality of the active route finder was created and integrated with the stage 1 graphical user interface, functionality included the nodeList, idList and left click events for the user to interact with the map overlay. Furthermore external files were also created and managed for storing data about the users route, as well as this accessibility settings such as the scaling and the measurement units  were integrated via the widgets and procedures.

**Test that have been carried out:**

Testing was carried out on the lists that contain the node and unique identifier data to ensure that they were appending the correct data and the indexes for distance calculations and such were accurate so that a distance for the journey could be calculated and given to the user.
Furthermore testing on the data inputs for the fuel prices and their subsequent calculations were also carried out as they must be validated by the software, as a result a valid input must be put in by the user to continue with the calculations.

**How it meets success criteria and user expectations:**

- Functions to start and end the journey (this section was adapted so that instead of user inputs of coordinates determining the start and end coordinates, now the user starts placing from an initial node and once they press the end route button the route is finalised.)
- Accurate overlay representation (user inputs directly onto the map overlay ensures that the nodes will be placed accurately and to the users requirements)
- Accurate and up to date global map (tkintermapview is installed with the latest openstreetmap tiles)

- Concise statistics and data representation (the statistics tab was switched to the data tab and instead of initially showing graphs now only numerical data is available for the user to access)

**Evidence of key stages that've been implemented:**

Through the functionality section of the active route finder, most of the code for the functionality was created in procedures and then implemented into the GUI through commands on the GUI widgets.

```
def changemode(self, newmode: str):
    ctk.set_appearance_mode(newmode)

#creating functionality for the user interface scale customiser

def differscale(self, newScale: str):
    newScale_float = int(newScale.replace('%', '')) / 100
    ctk.set_widget_scaling(newScale_float)
```

```
def leftClick(self, coordinates_tuple):
```

```
def pathConnect(self):
    pathList.append(currentNode)
    print(pathList)

def clearLastNode(self):
  #nodeList.pop()
  #idList.pop()
  #totalList.pop()
  pass
```

This is a collection of a couple of the functionality procedures that were implemented into the active route finder through commands.
Using procedures and subroutines allows the code to be reused and made into new subroutines that can have similar functionality or structure but allows an improvement in coding efficiency, this has allowed the code to become smaller and more efficient along the development.

Comments have also been added through the subroutines in order to improve code readability and maintainability for future development, by annotating what a certain script's function is or outlining a subroutines process.

In the functionality section multiple data structures have been used such as lists, tuples and strings.

```
global pathList ; pathList = []
global nodeList ; nodeList = []
global idList ; idList = []
global positionList ; positionList = []
global totalList ; totalList = []
global distList ; distList = []
```

```
def leftClick(self, coordinates_tuple):

    global currentNode ; currentNode = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
    nodeList.append(currentNode)
```

Lists have been accommodated to store coordinate and identifier values for waypoints that have been plotted onto the map overlay by the user, then the tuples were used in order to concatenate the two lists into one list where each index contains a corresponding identifier and coordinate value.

Error handling was used alongside the validation subroutines to validate the user inputs that were inputted into the data tab on the active route finder GUI.

```
def validations(self):

    try:
        inputData = float(self.fuelEntry.get())
        print(len(str(inputData)))

        if len(str(inputData)) < 6:
            print('input too short')
        if len(str(inputData)) > 6:
            print('input too long')

    except:
        print('invalid fuel price input')
```

This validation subroutine included try/except statements that accepted inputs and then format checked them, should an error occur the try/except statements were included in order to make sure the program would not crash and would instead print warnings, mainly to the command line interface, but also to the active route finder so the users would know what the issue is.

**Changes in design of software in stage 2:**

Unfortunately the machine learning algorithm that was initially proposed to be a part of the program could not be added, this was due to limitations occurring from the tkintermapview module as well as memory management.
As the design progressed and research was initiated I realised there was not enough documentation and data available on the tkintermapview module in order to link the placement of nodes to a suitable machine learning algorithm to be set upon, furthermore the machine learning algorithm would have consumed large amounts of memory and would have most likely had a negative impact on the performance of the active route finder software.

**Summary of whole software as an iteration of a prototype at this stage:**

The software is now nearing the maintenance stage of its product development lifecycle, so far the core functionality and usability of the software has been maintained as well as some precautionary testing and bug fixing in order to create an iteration that functions as the user expects it to.
After this stage the final testing and maintenance must be carried out to remove any undetected bugs from the software as well as improve usability of the software where it can happen.

# Post-Development Testing:

In this section the full release version of the active route finder will be tested to ensure that all functionality and usability features are working as expected and testing the system as a whole in order to observe the connections between different components of the software and their behaviour when interacting with each other when the software is running.

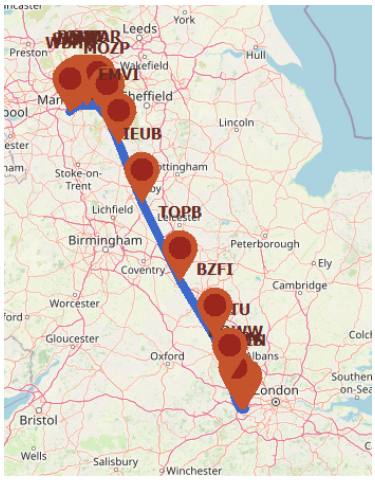
**Post-Testing for Function and Robustness:**

We are now going to test the robustness and functionality of the software in terms of the stakeholder requirements and success criteria that were identified in the analysis section.
First of all we will be plotting simple waypoints along the map overlay, we'll be adding a large amount of waypoints to see that the software can handle large amounts of data input and still perform as expected without any issues.

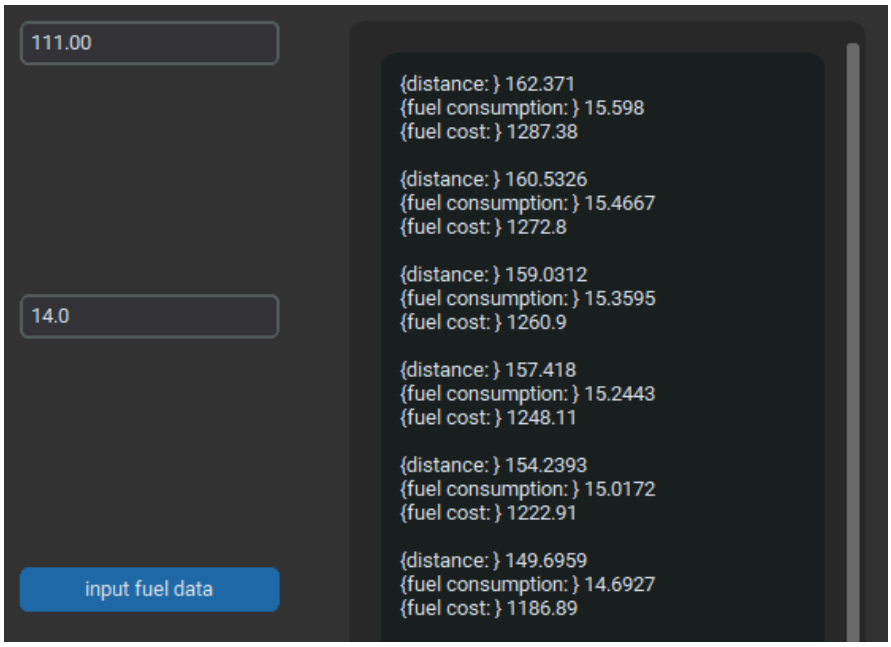
Next we'll be adding real life consumption and price values to the input boxes in order to view the increment of the values as more and more waypoints are added to the map overlay.
And to create a lifelike scenario we will be creating a route from London to Manchester via aeroplane.

**Input Variables:**

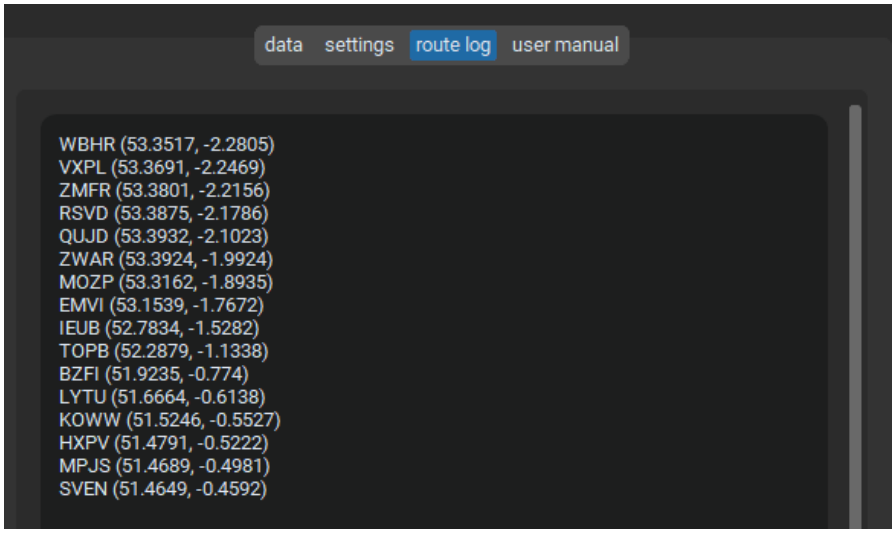
| Fuel Price per Litre (Jet A1 Fuel) | Cessna 172 MPG Rate |
|---|---|
| 111.00 | 14.0 |




Here is the route that was made from London Heathrow Airport to Manchester Airport



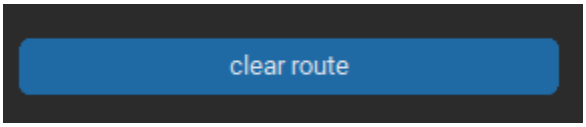
Here are the fuel consumption, distance and fuel cost variables produced, by using around 15 hrs worth of aviation fuel it would cost around £1287 to fly a distance of 162 miles from London to Manchester.

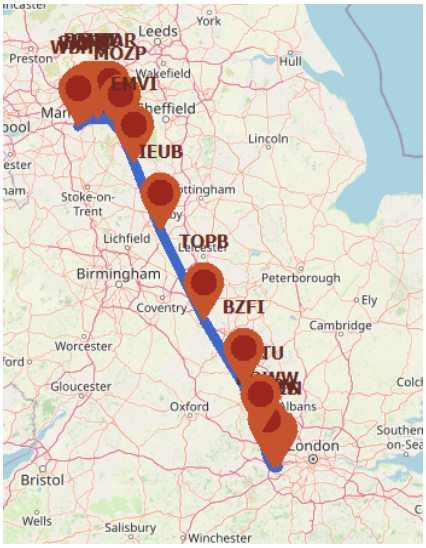


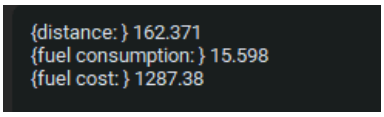
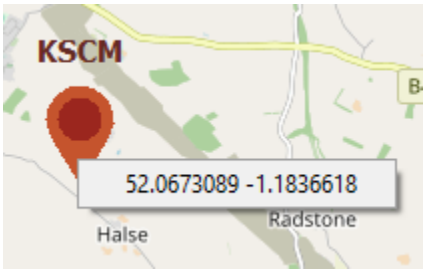

This is the route log of the nodes placed from the journey of London to Manchester.

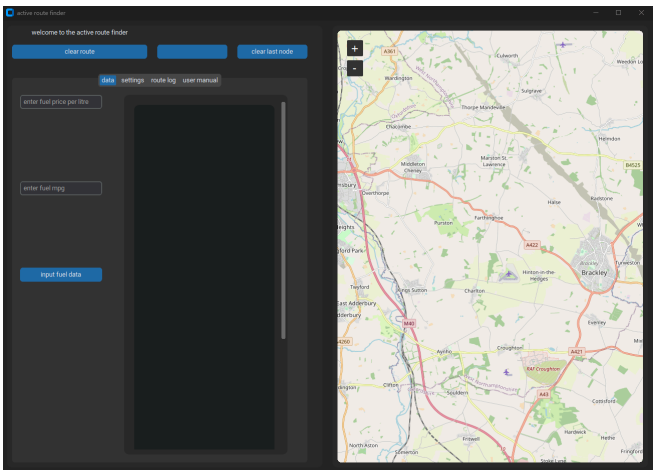
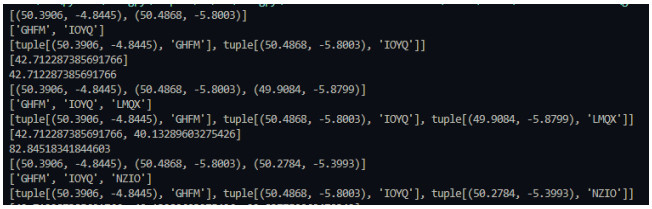
**Cross-reference of success criteria met from requirements and testing:**

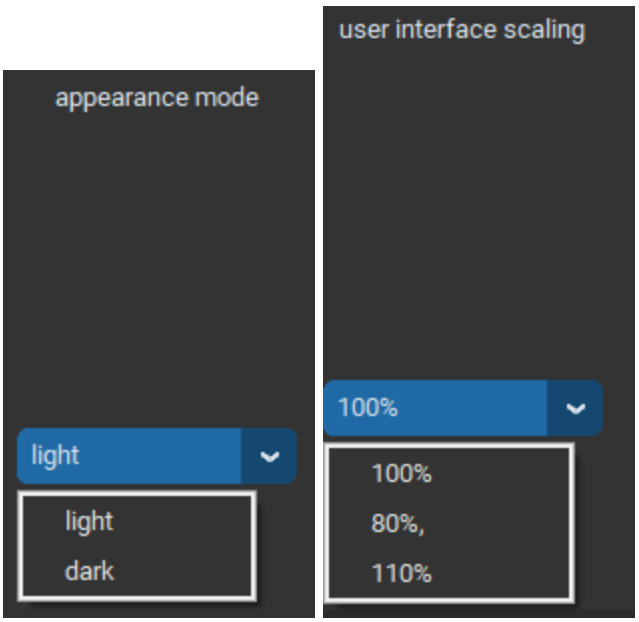
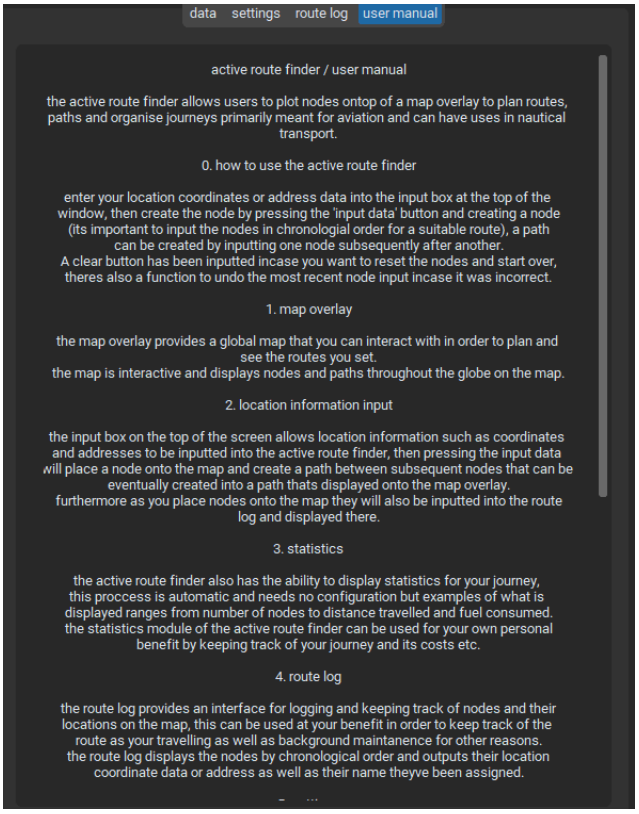
The testing for the ease of accessibility was carried out by the stakeholders post development testing on page 120 where constructive criticism and positive feedback was received on the final iteration of the software.

| **Success Criteria / Requirements** | **Evidence** |
|---|---|
| Functions to start and end journey |  This is the button used to clear the route of waypoints from the overlay, its functionality allows the waypoints to be also cleared from the text file and the route log |

| | |
|---|---|
| Accurate overlay representation of journey<br>Accurate node waypoint information |  <br>WBHR (53.3517, -2.2805)<br>VXPL (53.3691, -2.2469)<br>ZMFR (53.3801, -2.2156)<br>RSVD (53.3875, -2.1786)<br>QUJD (53.3932, -2.1023)<br>ZWAR (53.3924, -1.9924)<br>MOZP (53.3162, -1.8935)<br>EMVI (53.1539, -1.7672)<br>IEUB (52.7834, -1.5282)<br>TOPB (52.2879, -1.1338)<br>BZFI (51.9235, -0.774)<br>LYTU (51.6664, -0.6138)<br>KOWW (51.5246, -0.5527)<br>HXPV (51.4791, -0.5222)<br>MPJS (51.4689, -0.4981)<br>SVEN (51.4649, -0.4592)<br>Here is a route from London to Manchester the waypoints were entered manually and follow a course over airspaces between the two cities, since the waypoints were placed manually they are likely to be accurate and therefore the coordinates that were pulled from the waypoints are fully representative and accurate. |

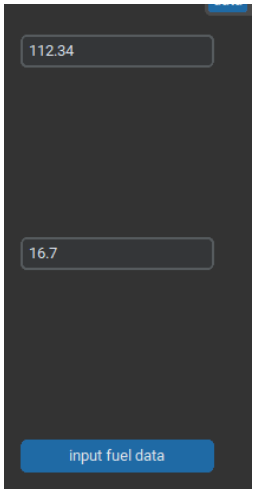
| | |
|---|---|
| Accurate and up to date global air network map |  <br>The OpenStreetMap tiles that are used in the tkintermapview module are fully up to date and include borders, location names and road and transport networks as well as airports marked on the map. |
| Concise statistics and data representation |  <br>Here is a test output from the calculations that were inputted from the earlier route, the distance, consumption and cost variables are all within the expected range for the calculations. |
| Hovering or clicking on a node reveals it waypoint information |  <br>This occurs when the user left clicks the node.<br>The user is able to view the coordinate data and then click on it to copy it for later use. |

| | |
|---|---|
| Easy to understand user interface |  <br>Simple to understand and modern GUI |
| Appropriate data handling |  <br>Here the data from the map overlay has been organised and appended to lists and outputted to the command line interface for testing purposes. However data is stored in separate sections depending on its usage and type. Also data is held in the external text file for the route itself without any lists. |

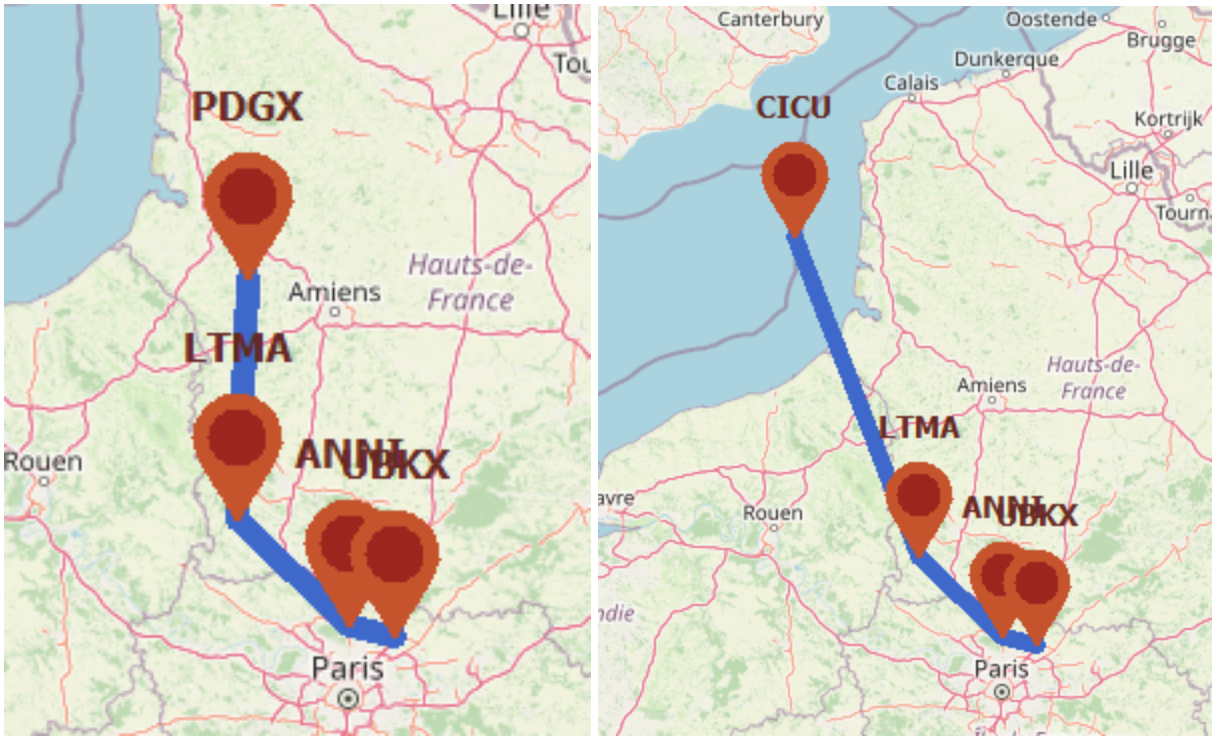
| | |
|---|---|
| Customisations for accessibility |  <br>Here are the user accessibility functions which include user interface scaling and themes for the software, these additions allow for the user to customise the software to meet their own needs. |
| User manual |  |

# Annotated Usability Testing:

For this section of testing we will be creating a real life scenario where the active route finder will have to be used and viewing the functionality of the software and how it performs under life-like conditions, this qualitative type testing allows us to gain information on whether the software will be usable by the stakeholders in the situations it is intended for.
The main stakeholder, Giancarlo intended to test the software for a route he planned from Paris to Edinburgh. Here are the records of Giancarlo's testing of the active route finder.



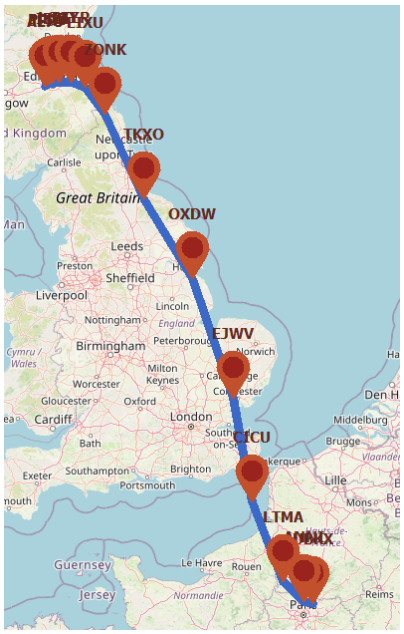

| Fuel price per litre | Aircraft fuel consumption rate |
|---|---|
| 112.34 | 16.7 |




On the left is an image of Giancarlo's original route, but due to an incorrect heading he replaced it with this new route on the right using the clear last node button.

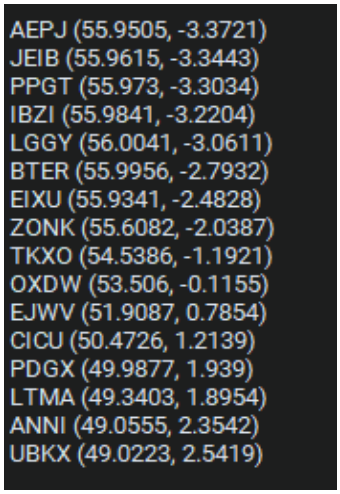
Giancarlo's response:

"The clear last button was helpful in removing some of my misplaced nodes that I accidentally added to the path, however one issue with the clear last button is that sometimes it will remove the node completely but the original path to that node may still remain for some time disappearing, creating a confusing looking route."

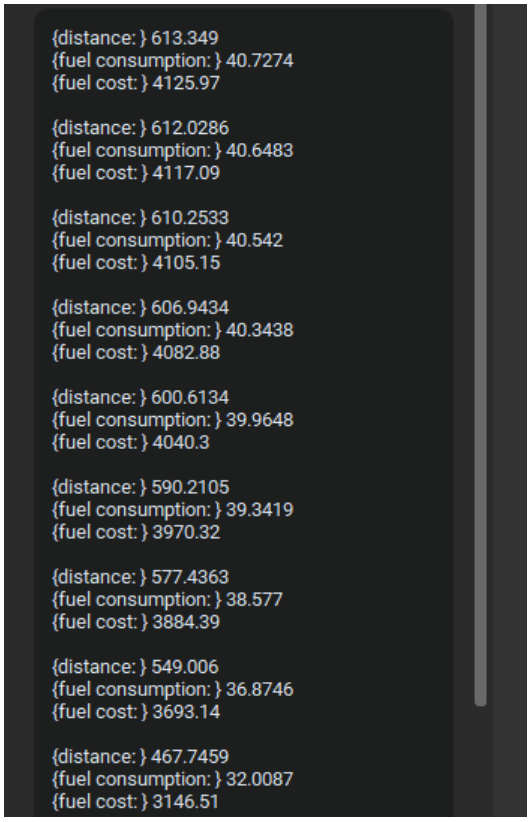



Giancarlo's response to his route creation:

"The route was designed and created by me so there was little room for error, and using the clear last waypoint button made it easier to plot routes with waypoints without having to start a new route if I had incorrectly placed a waypoint"




Here is the route log produced by Giancarlo's journey.




Giancarlo's response to the data section:

"Using manual inputs for the fuel consumption rate and fuel price for the route were tedious at times but the information, even though an estimate, was good enough to give me a rough guess of costs and calculations that were involved for logistics of sourcing materials such as fuel and funds for it.
One piece of criticism I would like to put out about this section is the lack of units that were present which made reading the values at times a bit disorientating."

Giancarlo's response to the other features of the active route finder:

"I had the opportunity to explore the active route finder and the first section I'd like to discuss is the user manual, it was helpful at first acclimating to a new environment and gave useful advice on how to use the software to create routes and what other functionality it contained,

The settings tab was one of my personal favourite features due to the amount of customisation available from the distance units to even the theme and scaling of the software was very helpful to adapt the software to my needs and my specific device.
Features like the route log also provided great insight into my journey, by providing me with the coords and waypoint names for the journey in chronological order was useful to have at hand when tweaking the route, however it would be helpful for removed waypoints to be also removed from the route log when a journey ends for example, otherwise a long list of nodes from multiple journeys accumulate and that can become confusing for me."

**Evaluating Giancarlo's Feedback:**

Giancarlo gave positive statements for most of the route creation functionality of the active route finder, however the issue relating to path of deleted nodes remaining may relate to memory issues from specific devices so in development the specifications for future devices may have to be raised in order to avoid issues like this.
Some constructive criticism was received on the labelling of data in the data tab textbox which can be amended through future iterations by appending units for the data variables, furthermore the issue with the route log can be amended by adding a few lines of code that clear the textbox everytime the clear route button is pressed.
However on the flip side positive feedback was received on the user manual and settings tabs, meaning customisations and guidance are important to stakeholders, these can be expanded in future updates in order to contain more settings and tips that can aid the user and make it more adaptable.
Therefore we can conclude that the software has had a positive impact on the stakeholder Giancarlo even though some setbacks were found it seems that the software services the needs for private pilots that have a need for an open source and free routing software to aid them in their journeys.

# Evaluation:

## Success criteria met:

| **Test criteria** | **Results** |
|---|---|
| The functionality of the start/stop journey function | Functionality was successfully met as the software supports the feature to stop the route as well as automatically start a new one. The user is in control of the stopping of the route improving the user experience. |

| | |
|---|---|
| | Evidence for this criteria is available on pages 93-94, the test was partially tested and was not able to be fully completed due to an issue with the route log removing data correctly. |
| Graphical map overlay with accurate waypoint and route display | This criteria was successfully met as the map overlay was inherited from OpenStreetMap tiles through tkintermapview ensuring up-to-date maps were included, the testing for the map overlay functionality is available on pages 99, 95, 91-92 and 103. |
| Accurate waypoint information (latitude, longitude, altitude, unique identifier) | Waypoint latitude, longitude and identifiers were successfully shown, therefore successfully meeting the success criteria and test criteria the testing for the functionality was carried out on pages 73, 76, 81, 82, 85 and 100.<br>Through these test functionality was thoroughly tested and improvements iterated. |
| Accurate imported map with key information | Successfully shown up to date openstreetmap tile, testing was completed on page 60, an accurate openstreetmap supported by tkintermapview fully met the success and test criteria |
| Accurate representation of statistics and data from route navigation and vehicle. | Successfully displayed route data to the users from inputted data.<br>Validation tests numbered 1 through 3 on page 97 tested sanitisation and validity of inputted data through various data types including erroneous, normal and boundary data. |
| Suitable machine learning algorithm supported by software and imported data sets to be trained upon | This success criteria was not able to be completed due to the software and hardware limitations from including a route finding and machine learning algorithm as well as having data to train the algorithm on and ensuring it was free from bias and discrimination. |

| | |
|---|---|
| Function that allows the user to view waypoint information when it is selected. | Users can view node coordinates and identifier information, therefore the success and test criteria has been met fully, the testing for this component was carried out on page 109 in the post development testing where the user can actively right click the node in order to gain its coordinates. |
| Ensuring the user interface is easy to navigate around and easily accessible and understandable. | The criteria was fully met as explained through the post development stakeholder testing, however there is still room for improvement as more accessibility features could be added in order to improve the ease of access. |
| Detailed user manual that accurately provides information on how to use the software and a troubleshooting guide. | The user manual criteria was successfully implemented as a tab on the user interface and provided detailed information to the users on how to access and use the active route finder for their route planning.<br>The testing for the active route finder user manual can be found on pages 111 and 65 |
| Provide customisation features for accessibility | This success criteria was successfully met, the active route finder provides a settings tab that has functions such as interface scaling and unit measurements on page 111 testing was provided and in the post development stakeholder feedback positive and constructive criticism was received. |

**Evidence:**

The active route finder provides a modern and easy to access user interface, buttons are highlighted blue and labelled improving the navigation of the software as well as the user experience.

Here the user manual is provided in a tab on the main screen, users can easily gain information about the active route finder and how to use the software, furthermore adding troubleshooting information along with the user manual in order to account for any potential future problems that may occur.

On the user interface the map overlay is directly accessible from the main window screen, users can easily add nodes directly onto the map as well as other functions such as clearing the route and ending the route planning in preparation for the journey.

The data section of the user interface allows users to access their route statistics including fuel cost, consumption and distance travelled, this information allows the user to log certain information about their route which they can use for their own purposes.

All of the user interface and functionality components have gone through their own variety of testing methods in order to view how they react under normal conditions, extreme conditions.
Different data types have been inputted into input boxes in order to test their data validation methods, as well as seeing if any specific data could cause the software to crash or return any unexpected errors.
Furthermore while functionality was being integrated to the user interface, functional tests were carried out to ensure the subroutines behaved as expected and error handling was executed properly.

The first success criteria that was met was an accurate overlay of the journey that was met by the map overlay display of all the waypoints from the departure location to the destination location along with the route log that provided information about every single waypoint placed onto the map.
The functions to end and start a journey were embedded into the map itself and to start a new journey all the user had to do was press the clear route button and then start placing new nodes.
Accurate waypoint information was provided in the form of unique identifiers assigned to the waypoints from London to Manchester which also included coordinate data from the map itself, therefore the user knew the exact location and identifier of each waypoint by simply checking the route log.
This success criteria was tested through the iterative development of the software and corrected by trial and error in areas such as placement, height and width, styling and interactivity.
Throughout the development of the user interface and in the post development tests the map overlay was able to correctly function by providing an interactable portal for route planning and waypoints.
Testing for this component was carried out on pages 99, 95, 91-92 and through post development testing on page 103. This testing allowed the iteration of the map overlay component to meet the success criteria in its entirety, however for future development more features could be added to improve the user experience.

For the second success criteria the objective was to provide accurate waypoint information, in this case the success criteria was fully met as when the user views or selects the waypoint as shown below the coordinates can be displayed to the user to copy and paste if needed, furthermore the waypoints unique identifier information is also show automatically on the active route finder so that the user can easily and quickly identify the waypoint as well as its position in the route through the route log.

This section of the success criteria was tested through the route log tab and the command line outputs that were created in the iterative development of the active route finder in the functionality section. On pages 73, 76, 81, 82, 85 and 100 the waypoint data system including waypoint coordinates, waypoint unique identifier and distances between waypoints were thoroughly tested and iterated allowing for an accurate representation of the waypoints to be outputted.

The third success criteria was to provide an accurate and up to date global map for the users interaction with the software, as shown below the success criteria has been fully met as the tkintermapview module uses a OpenStreetMap tile to display the map to the display.

### Use other tile servers

TkinterMapView uses OpenStreetMap tiles by default, but you can also change the tile server to every url that includes `{x} {y} {z}` coordinates. For example, you can use the standard Google Maps map style or Google Maps satellite images:

```
# example tile sever:
self.map_widget.set_tile_server("https://a.tile.openstreetmap.org/{z}/{x}/{y}.png")  # OpenStre
self.map_widget.set_tile_server("https://mt0.google.com/vt/lyrs=m&hl=en&x={x}&y={y}&z={z}&s=Ga'
self.map_widget.set_tile_server("https://mt0.google.com/vt/lyrs=s&hl=en&x={x}&y={y}&z={z}&s=Ga'
self.map_widget.set_tile_server("http://c.tile.stamen.com/watercolor/{z}/{x}/{y}.png")  # paint
self.map_widget.set_tile_server("http://a.tile.stamen.com/toner/{z}/{x}/{y}.png")  # black and
self.map_widget.set_tile_server("https://tiles.wmflabs.org/hikebike/{z}/{x}/{y}.png")  # detail
self.map_widget.set_tile_server("https://tiles.wmflabs.org/osm-no-labels/{z}/{x}/{y}.png")  # r
self.map_widget.set_tile_server("https://wmts.geo.admin.ch/1.0.0/ch.swisstopo.pixelkarte-farbe/

# example overlay tile server
self.map_widget.set_overlay_tile_server("http://tiles.openseamap.org/seamark//{z}/{x}/{y}.png")
self.map_widget.set_overlay_tile_server("http://a.tiles.openrailwaymap.org/standard/{z}/{x}/{y}
```

As shown here in the tkintermapview GitHub the tiles are loaded from external servers and can even be switched to other tile servers such as google maps, openseamaps and openrailwaymaps.
Testing for this component was completed on page 60, through iterations of the user interface finally allowing a suitable scale and placement of the map overlay to be admitted as well as ensuring that the openstreetmap tiles were functioning correctly.

The fourth success criteria included is the customisations for accessibility, as shown in the images section below the success criteria was fully met as the active route finder included customizability for the user interface scaling, the measurement units, the theme of the user interface.
This clearly met the success criteria as the original success criteria was created in order to improve the efficiency of use by the user by including multiple accessibility and customization features in order to improve the user experience of the active route finder.

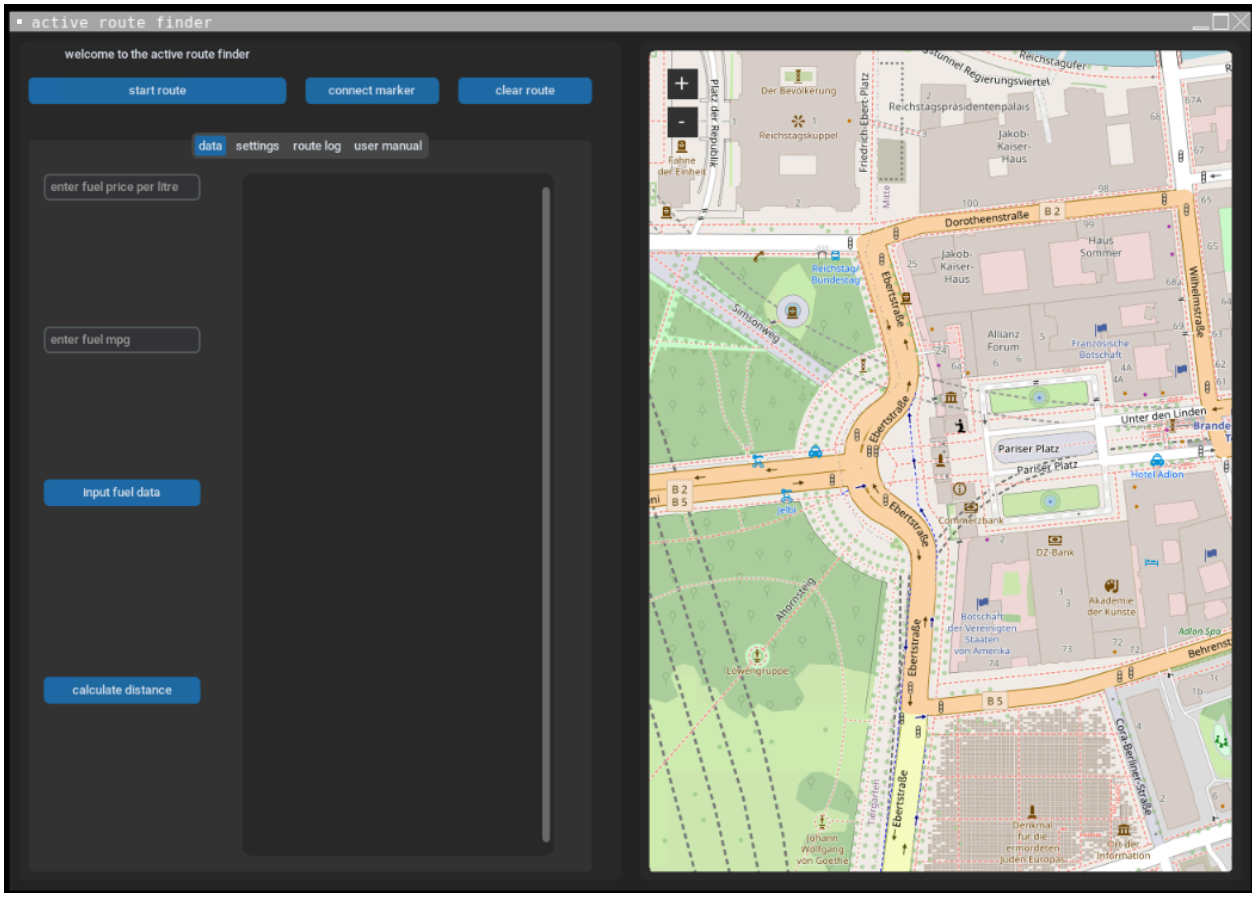
Another one of the success criterias for functionality was appropriate data handling, this was fully met as the software takes the data from the waypoints and transfers it to an external text file as well as the internal route log tab. As a result the data is appropriately handled and not introduced into any unnecessary steps that may decrease the efficiency and functionality of the software, furthermore the external text file has options to be reset through the clear route, or clear last node buttons this allows data to be erased from the files and the route log if the user does not want it to be there.
On pages 104, 100, 82 data handling between external files and the software itself was tested and results after iterations became positively aligned with the success criteria.

Concise statistics and data representation: the original success criteria outlined for a tab to include concise statistical representations of data as well as being able to carry out complex calculations for the inputted data from the user.
This success criteria for functionality was met in full as the data tab allows the user to input information with validations and insure that the data is sanitised, this allows the calculations be accurate and for the correct outputs to be given to the user, as if the outputs are inaccurate it could cause route issues that may be dangerous for the user.
Referring to the validation tests numbered 1 through 3 on page 97 we can see how boundary, normal and erroneous data was used to ensure data validation and sanitisation was implemented correctly.

Suitable machine learning algorithm with data to be trained upon was not met at all unfortunately.
The machine learning algorithm was not able to be met due to hardware and software limitations of the active route finder and the tkintermapview and customtkinter modules for the software.
Due to the tkintermapview module allowing waypoints to be displayed onto the overlay it was envisioned for the software to use Djikstra's algorithm in order to create a shortest viable route from a departure address to a destination location, however the memory needed in order to calculate the nodes and use machine learning in order to display it to the map overlay would have cause memory issues as the software and Python program may not have had enough memory to sufficiently run the program under all the load.
Furthermore, placing waypoints around the UK for example so that Djikstra's algorithm could work would have to incorporate some system in order to randomly or systematically place waypoints across the overlay.
As well as this the machine learning algorithm would take large amounts of data and power to train as a result the benefits are outweighed by the disadvantages and therefore it was not worthy to implement this machine learning algorithm in accordance with Djikstra's algorithm.
In conclusion the machine learning algorithm and Djikstra's algorithm were not suitable to be adapted and implemented into the active route finder, therefore their presence has to be redacted from the active route finder in order to maintain the simplicity and ease of accessibility for the software.

Another success criteria to be cross-referenced is the ease of usability for the user interface, the user interface has been made easy to use by ensuring the two main components of the program are readily

available from the main screen and that the every component is suitably labelled so that the user can identify components from each other in order to improve usability.




On the right hand side the map overlay is easily accessible and can be interacted with directly after opening the software, even if no waypoints or routes have been plotted yet.
On the left hand side the second component of interactivity is available which allows users to input data, change settings and manipulate paths and waypoints that have been added to the map overlay.
The overall accessibility and usability for the software in accordance to the success criteria has been fully met, however the usability can still be further improved in order to provide more functionality and settings in the settings tab, through future development this could include further accessibility functions like changing the background theme to different colours, including text to speech support for inclusivity, including variable map overlay size in order to improve usability.

## Usability features:

The dark mode theme was chosen as default along with the blue colour scheme for widgets so that they can contrast from the frames and provide clear indicators for users to what parts of the software provide interactions, buttons and option menus have been labelled clearly along with the tab view

sections in order to ensure that from the main starting screen that the user can see what widgets do what from a simple glance.

The map overlay provided by tkintermapview was intended to take up half of the window, as well as allowing the user to expand the window to ensure that the map overlay was a major focus on the window, splitting down the functionality interactions and the map overlay half and half through the screen allows the user to have easy access and view to the map overlay and widget interactions on the same screen to allow for multitasking as well as improving the usability and user experience.

# Limitations:

As it was initially intended the active route finder was supposed to include a machine learning aspect as well as Djikstra's algorithm to the map overlay in order to allow improved efficiency in the use of the program. However this was switched out to allow the user to manually place nodes onto the map instead of placing one start and one end node, this was done mostly due to hardware and software limitations which would have placed great strain on the software to accommodate these algorithms into the software.
The amount of processing power used to train and output the machine learning algorithm onto the map overlay as well as the inefficiencies of Djikstra's algorithm made it unsuitable for the software.

**How these limitations were overcome:**

Instead of using complex algorithms, the development was changed in order to allow more customisation on the placement of nodes for the users.
Also the active route finders input fields have limitations as if the user does not know specific information such as the fuel price by litre or the fuel consumption of their mode of transport then they will not be able to access the fuel consumption and cost data.
Furthermore porting the software will be required in some instances to use the software as running the software on a laptop in all situations may not be suitable for the circumstances, furthermore there are connected issues with porting the software to other devices.

**How to avoid these limitations:**

Due to the limitations of the entry fields some error handling was implemented into the entry fields so that if there was no entry given to the fields then the software will not display and print N/A to the label instead of causing the program to crash, which would lead to inefficiencies in the program when in use.
However the limitation of the software being ported to other devices and potentially not being ready to run on them is a different type of limitation as the solution would require lots of integrating with

different operating systems and device architectures that could be achieved in the future but not in this release iteration of  the active route finder.

## Maintenance:

Maintenance of the software has been improved for future development by including extensive commenting of the procedures and class, as well as making sure that variable names have been assigned accordingly so that from a developer perspective it is easier to identify the connections between components in the subroutines and their purpose.

Some potential maintenance issues that could occur in the future coincide with widget placements on the frames, even though the frame propagate state are set to false, placing future widgets for expansion may change the placement settings of other widgets even if it doesn't effect the frame the widgets are on. This could cause future issues for the expansion of the software in future updates, to fix this any widgets added would have to be observed on the software to see their impact on surrounding widgets and then placed accordingly and potentially being limited to where and what can be placed. More limitations that can occur with the maintenance process over the development lifetime of the software include issues with improving the waypoint system and adding improvements to that system, because the waypoint system is a complex web of connected lists, concatenation and manipulation. Adding more functionality to the waypoint system would need careful consideration on the developer side as one change to an aspect of the system could cause lasting consequences on different components such as the file writing system, the distance and fuel calculations and even the displaying of waypoints on the map overlay.

## Further Development:

In the future in terms of maintenance and future development and additions to the software, functionalities such as introducing a subtle assistance algorithm which provides multiple potential next nodes the user could input could be added to improve the functionality and the user experience of the program.
Furthermore the user experience could be improved and expanded by allowing users to have access to readily available popular locations or have them on a selectable dropdown box in order to easily locate the locations and improve the user experience.

**Unmet criteria development:**

The components that were unmet in the success criteria for the active route finder included a successful working version of Djikstra’s algorithm and machine learning to support the process, however to try and attain these components in the future the Djikstra’s algorithm could be improved

to the A* algorithm instead which would implement a more efficient pathfinding algorithm that would reduce memory usage and intended rendering time.
The A* algorithm is different from Djikstra's algorithm in the way it includes heuristic cost values as well as total cost values for pathfinding, therefore the search from one node to another can be more streamlined and direct than Djikstra's.
To implement the new algorithm alongside machine learning in future development would require a large data set to train the machine learning algorithm in order to create a neural network that can identify suitable waypoints and where to place them for maximum efficiency in the route, or else a machine learning module with specific pre downloaded data sets can be used in forms of modules or API's to provide a framework for the pathfinding algorithm to draw from.
Some unmet features also include the selection options for specific aircraft types and live fuel consumption API's so that users do not have to manually input fuel and consumption data into the software which may become tedious over time.
This was unable to be met due the initially planned module to be used was only available on a more outdated version of Python which would cause conflicts with other components of the software.
A work around for this in the future could include manually creating a module filled with data about different vehicle and aircraft types as well as integrated fuel consumption API's embedded, or to research to find a newer module that does these processes but also supports a newer version of Python.

**Partially met usability features:**

Some usability features that were not met in full include the initial plan to include data visualisations for data, this was met in the sense that users were able to view consumption and route data in figures but not in graph form, this could be added through further iterations through seaborn or matplotlib.
Another partially met usability feature included the node information access via clicking on the waypoint on the map overlay, the only data available to the user is coordinate data, the initial plan was to include further information such as distance from and to next and previous nodes, however due to the tkintermapview modules structuring it was not possible to attain.
Some more partially met success criteria include the accessibility and customisations of the user interface

# Stakeholder feedback and testing:

A pre-release version of the software was provided for the stakeholder from the development section in order to gain their feedback on the final iteration of the program.

Here are some questions that were directed for their feedback:

- Are you satisfied with the final version of the active route finder?
- Is the active route finder easy for you to access and use?
- Did the active route finder aid you in your transport?
- Were the additional settings helpful?

Here are the responses from the Pilot we mentioned earlier, Giancarlo.

**Are you satisfied with the final version of the active route finder?**
**“**The active route finder proved to be a valuable resource to me as a private pilot, though it may have not provided as many features that mainstream pilots can access it still managed to provide a stable and easy to use interface, from which I was able to plan and alter my routes from.**”**

**Is the active route finder easy for you to access and use?**
**“**In short, yes. The softwares interface was managed very well and easy to understand through the use of the user manual, so that I found it no issue to quickly learn how to navigate the interface. Furthermore I could critique some of the scaling for the software, on aircraft we could be using many different devices so providing more scaling options would have helped the software adapt to its environment a bit better.**”**

**Did the active route finder aid you in your transport?**
**“**Yes, the interactive map made it quite easy to directly select an area on the map and add a waypoint to that location and to create paths between waypoints. As well as this the data for fuel costs and consumptions were useful for tracking my fuel expenditure as well as extrapolating how much estimated fuel I should use on future trips.**”**

**Were the additional settings helpful?**
**“**Most of the settings in the settings tab, such as the unit measurement selector and theme were useful in customising the route specifications, as well as the clear route button. However an issue was the fact that creating a waypoint in the wrong location was eventful as to correct the route you would have to reset the whole route instead of being able to remove the most recent node.**”**

**Stakeholder Feedback Analysis:**
It seems that overall Giancarlo made good use of the active route finder, however there were some areas of improvement that were identified through this feedback that could be potentially added through future development and maintenance, this includes some user functionality and some quality of life improvements.

# Final Program Files:

## Main Python File:

```python
import customtkinter as ctk
import tkinter as tk
import tkintermapview as tmv
import numpy as np
from geopy import distance
import random
import string

#presetting window theme and mode

ctk.set_default_color_theme('blue')
ctk.set_appearance_mode('dark')

class arf(ctk.CTk):
    def __init__(self, title, size):
        super().__init__()

        #creating presets for window size and title

        self.title(title)
        self.minsize(size[0], size[1])
        #self.maxsize(size[0], size[1])
        self.geometry(f'{size[0]}x{size[1]}')

        #creating grid for widgets

        self.grid_rowconfigure((0,1,2), weight=1)
        self.grid_columnconfigure((0,1), weight=1)

        #create frame for menu buttons

        self.barFrame = ctk.CTkFrame(self, corner_radius=10)
        self.barFrame.grid(row=0, column=0, columnspan=1, rowspan=3, padx=10, pady=10, sticky='nsew')
        self.barFrame.grid_columnconfigure(4, weight=1)
        self.barFrame.grid_rowconfigure(2, weight=1)
        self.barFrame.grid_propagate(False)
```

```
37
38          #add widgets to barFrame
39
40          self.textLabel = ctk.CTkLabel(self.barFrame, text='welcome to the active route finder')
41          self.textLabel.grid(row=0, column=0, padx=10)
42
43          self.clearButton = ctk.CTkButton(self.barFrame, text='clear route', width=270, command = self.clearInput)
44          self.clearButton.grid(row=1, column=0, padx=(10), pady=10)
45
46          self.clearLast = ctk.CTkButton(self.barFrame, text='clear last node', command = self.clearLastNode)
47          self.clearLast.grid(row=1, column=2, padx=10, pady=10, sticky='e')
48
49          self.placeholderButton = ctk.CTkButton(self.barFrame, text='')
50          self.placeholderButton.grid(row=1, column=1, padx=10, pady=10, sticky='e')
51          self.placeholderButton.configure(state='disabled')
52
53          #create a frame for the tabview and configure it
54
55          self.multiBox = ctk.CTkTabview(self.barFrame)
56          self.multiBox.grid(row=2, column=0, columnspan=3, padx=10, pady=10, sticky='nsew')
57          self.multiBox.add('data')
58          self.multiBox.add('settings')
59          self.multiBox.add('route log')
60          self.multiBox.add('user manual')
61          self.multiBox.set('data')
62
63          self.multiBox.tab('user manual').grid_columnconfigure(1, weight=1)
64          self.multiBox.tab('user manual').grid_rowconfigure(1, weight=1)
65
66          self.multiBox.tab('settings').grid_columnconfigure((0,1), weight=1)
67          self.multiBox.tab('settings').grid_rowconfigure((0,1,2,3), weight=1)
68
69          self.multiBox.tab('data').grid_columnconfigure((0,1,2), weight=1)
70          self.multiBox.tab('data').grid_rowconfigure((0,1,2), weight=1)
71
72          self.multiBox.tab('route log').grid_columnconfigure(1, weight=1)
```

```
        self.multiBox.tab('route log').grid_rowconfigure(1, weight=1)

        #create widgets for the settings tab

        self.appeareanceModeText = ctk.CTkLabel(self.multiBox.tab('settings'), text='appearance mode', anchor='w')
        self.appeareanceModeText.grid(row=0, column=0, padx=5, pady=5)

        self.appeareanceModeOption = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values=['light', 'dark'], command=self.changemode)
        self.appeareanceModeOption.grid(row=1, column=0, padx=5, pady=5)

        self.scalingLabel = ctk.CTkLabel(self.multiBox.tab('settings'), text='user interface scaling', anchor='w')
        self.scalingLabel.grid(row=2, column=0, padx=5, pady=5)

        self.scalingMenu = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values=['100%', '80%,', '110%'])
        self.scalingMenu.grid(row=3, column=0, padx=5, pady=5)

        self.unitLabel = ctk.CTkLabel(self.multiBox.tab('settings'), text='configure unit measurements')
        self.unitLabel.grid(row=2, column=1, padx=10, pady=10)

        self.unitOpt = ctk.CTkOptionMenu(self.multiBox.tab('settings'), values=['miles', 'kilometres'])
        self.unitOpt.grid(row=3, column=1, padx=10, pady=10)

        #create frame and add widgets for the data tab

        self.fuelEntry = ctk.CTkEntry(self.multiBox.tab('data'), placeholder_text='enter fuel price per litre')
        self.fuelEntry.grid(row=1, column=0, padx=10, pady=10, sticky='new')

        self.consumptionEntry = ctk.CTkEntry(self.multiBox.tab('data'), placeholder_text='enter fuel mpg')
        self.consumptionEntry.grid(row=1, column=0, padx=10, pady=10, sticky='ew')

        self.fuelEntryButton = ctk.CTkButton(self.multiBox.tab('data'), command = self.validations, text='input fuel data')
        self.fuelEntryButton.grid(row=1, column=0, padx=10, pady=10, sticky='sew')

        self.dataBox = ctk.CTkScrollableFrame(self.multiBox.tab('data'), corner_radius=10, height=700, width=300)
        self.dataBox.grid(row=1, column=1, rowspan=2, padx=10, pady=10, sticky='nse')
        self.dataBox.grid_columnconfigure(1, weight=1)
        self.dataBox.grid_rowconfigure(1, weight=1)
```

```

        self.dataTextbox = ctk.CTkTextbox(self.dataBox, corner_radius=10, height=1000)
        self.dataTextbox.grid(column=1, row=1, padx=10, pady=10, sticky='nsew')

        #create and add widgets to the usermanual tab

        self.scrollable = ctk.CTkScrollableFrame(self.multiBox.tab('user manual'))
        self.scrollable.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
        self.scrollable.grid_columnconfigure(1, weight=1)
        self.scrollable.grid_rowconfigure(1, weight=1)

        self.manualText = ctk.CTkLabel(self.scrollable, corner_radius=10, height=1000)
        self.manualText.grid(column=1, row=1, columnspan=1, padx=10, pady=10, sticky='nsew')


        #lines 104 to 108 open an external text file in order to read the user manual and place it into the user manual tab

        f = open('arfusermanual.txt', 'r')
        textFile = f.read()
        f.close()

        self.manualText.configure(text=textFile)

        #creating new infrastructure for a textbox instead of label and scrollable frame so data can be inputted into route log easier

        self.scrollableLog = ctk.CTkScrollableFrame(self.multiBox.tab('route log'))
        self.scrollableLog.grid(row=1, column=1, padx=10, pady=10, sticky='nsew')
        self.scrollableLog.grid_columnconfigure(1, weight=1)
        self.scrollableLog.grid_rowconfigure(1, weight=1)

        self.logText = ctk.CTkTextbox(self.scrollableLog, corner_radius=10, height=1000)
        self.logText.grid(row=1, column=1, columnspan=1, padx=10, pady=10, sticky='nsew')
        self.logText.configure(state='normal')

        #create a frame for the map overlay

```

```
146             self.displayFrame = ctk.CTkFrame(self, corner_radius=10)
147             self.displayFrame.grid(row=0, column=1, columnspan=1, rowspan=3, padx=10, pady=10, sticky='nsew')
148             self.displayFrame.grid_columnconfigure(1, weight=1)
149             self.displayFrame.grid_rowconfigure(1, weight=1)
150             self.displayFrame.grid_propagate(False)
151 
152             #add overlay to overlay frame
153 
154             self.overlay = tmv.TkinterMapView(self.displayFrame, corner_radius=10)
155             self.overlay.place()
156             self.overlay.grid(column=1, row=1, columnspan=1, rowspan=1, padx=10, pady=10, sticky='nsew')
157 
158             self.overlay.add_left_click_map_command(self.leftClick)
159 
160             #creating lists with the global scope in order to be used in the route procedures
161 
162             global pathList ; pathList = []
163             global nodeList ; nodeList = []
164             global idList ; idList = []
165             global positionList ; positionList = []
166             global totalList ; totalList = []
167             global distList ; distList = []
168             global fuelConnList ; fuelConnList = []
169             global fuelCostList ; fuelCostList = []
170 
171         #creating functionality for the change theme button
172 
173         def changemode(self, newmode: str):
174             ctk.set_appearance_mode(newmode)
175 
176         #creating functionality for the user interface scale customiser
177 
178         def differscale(self, newScale: str):
179             newScale_float = int(newScale.replace('%', '')) / 100
180             ctk.set_widget_scaling(newScale_float)
181 
182         #creating functionality for the left click events for map overlay
183 
184         def leftClick(self, coordinates_tuple):
185 
186             global currentNode ; currentNode = (round(coordinates_tuple[0], 4), round(coordinates_tuple[1], 4))
187             nodeList.append(currentNode)
```

```
188
189         for i in range(1):
190             global identifier ; identifier = ''.join(random.choices(string.ascii_uppercase, k=4))
191             idList.append(identifier)
192
193         totalList.append(tuple[nodeList[-1], idList[-1]])
194
195         #create marker from the selected coordinates from the left click event on map overlay
196         global marker ; marker = self.overlay.set_marker(coordinates_tuple[0], coordinates_tuple[1], text=identifier)
197         marker
198
199         #simulateanously open text file and write coordinate data and unique identifiers to that text file
200         fileCommand = open('arfrouteinfo.txt', 'a')
201         fileCommand.write(str(currentNode))
202         fileCommand.write(identifier)
203         fileCommand.write('\n')
204         fileCommand.close()
205
206         fileRead = open('arfrouteinfo.txt', 'r')
207
208         #integrate a running total for distance in left click event
209
210         if len(nodeList) > 1:
211             for x in range(1):
212                 if self.unitOpt == 'kilometers':
213                     dist = distance.distance(nodeList[-1], nodeList[-2]).kilometers
214                     distList.append(dist)
215                     if len(distList) > 0:
216                         totalDistance = sum(distList)
217                         print(nodeList)
218                         print(idList)
219                         print(totalList)
220                         print(distList)
221                         print(totalDistance)
222
223                         # fuel consumption/fuel cost calculations
224
225                         if self.consumptionEntry.get() != '' and self.fuelEntry.get() != '':
226                            fuelMpg = self.consumptionEntry.get()
227                            fuelPrice = self.fuelEntry.get()
228
229                            #variables for the consumption and cost calculations
```

```
230
231                         fuelConn = float(dist) / float(fuelMpg)
232                         fuelCost = float(fuelPrice) * float(fuelConn)
233
234                         round(fuelConn, 4)
235                         round(fuelCost, 2)
236
237                         #simple list manipulation for consumption and cost lists
238
239                         fuelConnList.append(round(fuelConn,4))
240                         fuelCostList.append(round(fuelCost, 2))
241
242                         sumFuelCost = sum(fuelCostList)
243                         sumFuelConn = sum(fuelConnList, 4)
244
245                         roundSumCost = round(sumFuelCost, 2)
246                         roundSumConn = round(sumFuelConn, 4)
247                         roundDistance = round(totalDistance, 4)
248
249                         #converting numeric values into strings in order to be added to textbox
250
251                         costOutput = str('fuel cost: '), roundSumCost
252                         fuelConnOutput = str('fuel consumption: '), roundSumConn
253                         viewDistance = str('distance: '), roundDistance
254
255                         #appending sequences for textbox
256
257                         self.dataTextbox.insert('0.0', '\n')
258                         self.dataTextbox.insert('0.0', '\n')
259                         self.dataTextbox.insert('0.0', costOutput)
260                         self.dataTextbox.insert('0.0', '\n')
261                         self.dataTextbox.insert('0.0', fuelConnOutput)
262                         self.dataTextbox.insert('0.0', '\n')
263                         self.dataTextbox.insert('0.0', viewDistance)
264
265                 else:
266                     dist = distance.distance(nodeList[-1], nodeList[-2]).miles
267                     distList.append(dist)
268                     if len(distList) > 0:
```

```
                totalDistance = sum(distList)
                print(nodeList)
                print(idList)
                print(totalList)
                print(distList)
                print(totalDistance)

                #fuel cost/consumption calculations

                if self.consumptionEntry.get() != '' and self.fuelEntry.get() != '':
                    fuelMpg = self.consumptionEntry.get()


                    fuelPrice = self.fuelEntry.get()

                    #variables for the consumption and cost calculations

                    fuelConn = float(dist) / float(fuelMpg)
                    fuelCost = float(fuelPrice) * float(fuelConn)

                    round(fuelConn, 4)
                    round(fuelCost, 2)

                    #simple list manipulation for consumption and cost lists

                    fuelConnList.append(round(fuelConn,4))
                    fuelCostList.append(round(fuelCost, 2))

                    sumFuelCost = sum(fuelCostList)
                    sumFuelConn = sum(fuelConnList, 4)

                    roundSumCost = round(sumFuelCost, 2)
                    roundSumConn = round(sumFuelConn, 4)
                    roundDistance = round(totalDistance, 4)

                    #converting numeric values into strings in order to be added to textbox

                    costOutput = str('fuel cost: '), roundSumCost
                    fuelConnOutput = str('fuel consumption: '), roundSumConn
                    viewDistance = str('distance: '), roundDistance
```

```
309
310                         #appending sequences for textbox
311
312                         self.dataTextbox.insert('0.0', '\n')
313                         self.dataTextbox.insert('0.0', '\n')
314                         self.dataTextbox.insert('0.0', costOutput)
315                         self.dataTextbox.insert('0.0', '\n')
316                         self.dataTextbox.insert('0.0', fuelConnOutput)
317                         self.dataTextbox.insert('0.0', '\n')
318                         self.dataTextbox.insert('0.0', viewDistance)
319
320
321
322         #connect nodes with paths
323
324         if len(nodeList) > 1:
325             global path ; path = self.overlay.set_path(nodeList)
326             path
327
328        #logText inserts for coordinates and identifiers
329
330         self.logText.insert('0.0', '\n')
331         self.logText.insert('0.0',  str(nodeList[-1]))
332         self.logText.insert('0.0', ' ')
333         self.logText.insert('0.0', str(idList[-1]))
334
335
336     # this procedure ensures that when the 'clear' button is pressed that the routeinfo file and markers are also cleared.
337     def clearInput(self):
338         self.overlay.delete_all_path()
339         self.overlay.delete_all_marker()
340         with open('arfrouteinfo.txt', 'r+') as dataFile:
341             dataFile.truncate(0)
342
343     #create a validation proceduere that validates fuel consumption and fuel price inputs in order to keep accuracy of calculations
344
345     def validations(self):
346
347        #try/except statements for error handling
348
349         try:
350             inputData = float(self.fuelEntry.get())
```

```
            print(len(str(inputData)))

            if len(str(inputData)) < 6:
                print('input too short')
            if len(str(inputData)) > 6:
                print('input too long')

        except:
            print('invalid fuel price input')

        try:
            consumptionInput = float(self.consumptionEntry.get())
            print(len(str(consumptionInput)))

            if len(str(consumptionInput)) > 4:
                print('mpg figure too large')
            if len(str(consumptionInput)) < 4:
                print('mpg figure too small')
        except:
            print('invalid fuel consumption input')

    #create a procedure that connects waypoints with a path

    def pathConnect(self):
        pathList.append(currentNode)
        print(pathList)

    #subroutine for clearing last node inputted

    def clearLastNode(self):
      nodeList.pop()
      idList.pop()
      totalList.pop()
      marker.delete()
      path.delete()


#this command will make sure that the window updates as the user goes about accessing it as well as ensuring the size and title of window

if __name__ == '__main__':
    app = arf('active route finder', (1300,900))
    app.mainloop()
```

```

print('user has exited the active route finder')

#wiping the data from the text file after the program is terminated

with open('arfrouteinfo.txt', 'r+') as dataFile:
    dataFile.truncate(0)
```

**Auxiliary Text Files:**

arfusermanual.txt

```
active route finder / user manual

the active route finder allows users to plot nodes ontop of a map overlay to plan routes,
paths and organise journeys primarily meant for aviation and can have uses in nautical
transport.

0. how to use the active route finder

enter your location coordinates or address data into the input box at the top of the
window, then create the node by pressing the 'input data' button and creating a node
(its important to input the nodes in chronologial order for a suitable route), a path
can be created by inputting one node subsequently after another.
A clear button has been inputted incase you want to reset the nodes and start over,
theres also a function to undo the most recent node input incase it was incorrect.

1. map overlay

the map overlay provides a global map that you can interact with in order to plan and
see the routes you set.
the map is interactive and displays nodes and paths throughout the globe on the map.

2. location information input

the input box on the top of the screen allows location information such as coordinates
and addresses to be inputted into the active route finder, then pressing the input data
will place a node onto the map and create a path between subsequent nodes that can be
eventually created into a path thats displayed onto the map overlay.
furthermore as you place nodes onto the map they will also be inputted into the route
log and displayed there.

3. statistics

the active route finder also has the ability to display statistics for your journey,
this proccess is automatic and needs no configuration but examples of what is
displayed ranges from number of nodes to distance travelled and fuel consumed.
the statistics module of the active route finder can be used for your own personal
benefit by keeping track of your journey and its costs etc.

4. route log

the route log provides an interface for logging and keeping track of nodes and their
locations on the map, this can be used at your benefit in order to keep track of the
```

```
arfusermanual.txt
43   route as your travelling as well as background maintanence for other reasons.
44   the route log displays the nodes by chronological order and outputs their location
45   coordinate data or address as well as their name theyve been assigned.
46
47   5. settings
48
49   the settings will provide some auxilary functions for displaying the window, such as
50   the theme for the window as well as functionality to change the window size
51   to your specifications as well scaling customisations for the window
```

# Bibliography


https://customtkinter.tomschimansky.com/documentation/

-used for the development and research of the customtkinter user interface component

https://github.com/TomSchimansky/TkinterMapView

-used for the development and research of the map overlay component

https://wiki.openstreetmap.org/wiki/OSMPythonTools

-used during the research for the map overlay component

https://geopy.readthedocs.io/en/stable/#module-geopy.distance

- used for developing distance calculations for map overlay component

https://pypi.org/project/haversine/

-used for the early research and development of the distance calculations for the map overlay component