Gathering Atmospheric Data

Using an Unmanned Air Vehicle

Henry Miskin

March 27, 2014

Abstract

This report looked at energy based path planning for atmospheric data collection using unmanned air vehicles in a predetermined area, with particular consideration to the sample quality of data collected. Sampling plans were used in combination with route planning algorithms to produce optimal routes through a sample area. These routes were related to the energy consumed to produce a model that allows energy based path planning. The model for energy based path planning was found to be viable for certain plane configurations.

This report is submitted in partial fulfilment of the requirements for the Degree of Bachelor / Master of Engineering, Faculty of Engineering and the Environment, University of Southampton

Acknowledgements

I would like to thank my supervisor Andras Sobester for his help and support on this project, firstly by having the confidence in allowing me to explore in a direction that interested me and secondly by offering much needed support at many stages through this process. As a result I have found the work that I have done both interesting and rewarding. I am proud to put my name to this piece of work and know that this would not be the case without the support given.

Contents

Word Count: 9924

List of Figures

Abreviations

Abreviation	Name
DTSP	Dubins travelling salesman problem
TSP	Travelling salesman problem
UAV	Unmanned aerial vehicle

Nomeclature

Symbol	Name	Unit
A	Aspect Ratio	1
C_D	Coefficient of Drag	1
C_L	Coefficient of Lift	1
C_{D_0}	Coefficient of Zero Lift Drag	1
C_{D_i}	Coefficient of Induced Drag	1
L/D	Lift to Drag Ratio	1
R	Turning Radius	m
Re	Reynolds Number	1
S	Wing Planform Area	m^2
V	Flight Velocity	$m \cdot s^-1$
α	Angle of Attack	$degrees$
β	Energy Coefficient	1
γ	Energy Factor	N
μ	Air Viscosity	$Pa \cdot s$
ρ	Air Density	$kg \cdot m^-3$
b	Wing Span	m
c	Wing Chord	m
e	Oswald Factor	1
g	Acceleration due to Gravity	$m \cdot s^-2$
m	Mass	kg
x	Longitude Coordinate	m
y	Latitude Coordinate	m
z	Altitude Coordinate	m

1 Introduction

Unmanned aerial vehicles (UAVs) are used to collect atmospheric data, the flightpath taken by the UAV to collect this data defines where the atmospheric reading of interest is sampled. The locations at which atmospheric data is sampled determines the quality of model that can be produced from the readings. To ensure that the route taken by the UAV is optimal in terms of the quality of data collected, this report utilises optimal sampling plans to define the locations that the UAV should pass through. Then to ensure that the UAV is able to collect as much data as possible, the route through the sample plan is optimised to reduce the energy cost of the route. In this report atmospheric data refers to a meteorological reading to be collected.

1.1 Aims

The aim of this project is to research, design and implement a UAV path planner that minimises energy consumption while optimising spread and depth of data collection.

1.2 Objectives

The following objectives define the measurable goals of the project required to fully complete the aim. The objectives are listed in the order of completion, where each objective is a prerequisite to the next.

1. Design a model to calculate the energy costs of flight between two points in space. This model will output the predicted energy required from the work done against drag and the change in gravitational energy.
2. Implement an algorithm that uses the energy model to compare the cost of different routes to result in the least energy consumed.
3. Consider the flight characteristics of the UAV to produce a navigable path which visits the waypoints in order, as defined by the route to produce an accurate flight plan and energy prediction.
4. Fit the results of flight plans in different sample areas to the resulting energy consumed to produce a model that enables optimal path planning based on the area of interest and total UAV energy.

2 Background Material

2.1 Problem Outline

The troposphere is the lowest atmospheric layer and of significant interest to meteorological researchers as "almost all weather develops in the troposphere" [Geographic, 2013]. The section of the troposphere that is closest to the earth is the atmospheric boundary layer, in this layer the atmospheric conditions are affected by the surface of the earth. These effects mean that modelling this section is much more complicated than other layers of the atmosphere, therefore "the boundary layer is still not represented realistically" [Teixeira et al., 2008]. Developing a greater understanding and thus better modelling of the atmospheric boundary layer will improve the ability to predict weather patterns and pollutant dispersion, to name but two benefits.

To collect atmospheric data a number of approaches have been used historically, "many years before the use of radio-controlled aircraft, the collection of in-situ measurements was primarily done with balloons, towers, and tethersondes' [Bonin, 2011]. These options were limited as they were not able to move within the area of interest to build a model. UAVs operate with "reduced human risk, but also reduced weight and cost, increased endurance, and a vehicle design not limited by human physiology" [Pepper, 2012] in comparison to their manned counterparts. This means they provide a cheap and mobile data collection platform.

There are 5 main classes of UAV for different requirements [Sarris and ATLAS, 2001]; the class of UAV best suited to collecting atmospheric data of a limited area is the close range class. This class of UAV: "require minimum manpower, training, and logistics, and will be relatively inexpensive" [Intelligence et al., 1996]. These benefits allow a greater number of researchers to have access to mobile data collecting platforms. Most small UAVs are "not capable of reaching above 5,000ft [1524m]" [Weibel, 2005] with their maximum range being less than 10km [van Blyenburgh, 2000].

2.2 Literature Review

To achieve the objectives presented in this project pre-existing material on energy modelling, tour planning, path planning, energy routing and sampling plans has to be considered.

M. Price and Curran, 2006 consider the design parameters that affect aircraft performance and detail considerations to increase the efficiency. The L/D ratio is identified along with the weight as driving parameters for aircraft range. Asselin, 1997 however looks at the top level aerodynamic equations of flight to provide equations capable af defining energy consumption under different flight modes. Together they provide for

an overview of flight efficiency and basic equations to estimate this. Raymer, 2006 looks further into aerodynamic equations of flight and presents an estimate for the Oswald Factor for standard airframes which is required in a number of the aerodynamic equations.

Bigg et al., 1976,Held et al., 1984,De Berg et al., 2010 calculate minimum length tours of more than two points through applying the Travelling Salesperson Problem (TSP). The TSP is concerned with "finding the shortest path joining all of a finite set of points whose distances from each other are given" [Held et al., 1984]. The TSP problem considered in these papers uses the euclidean distance where the "euclidean distances satisfy the triangle inequality" [De Berg et al., 2010]; "The triangle inequality implies that no reasonable salesman would ever revisit the same city: instead of returning to a city, it is always cheaper to skip the city and to travel directly to the successor city" [De Berg et al., 2010]. Given the euclidean distances the solutions obtained do not correspond to the shortest path for a either a energy based TSP or a Dubins Travelling Salesman Problem (DTSP)

Dubins, 1957, Boissonnat et al., 1993 consider the shortest path of a vehicle with a bounded turning radius in two dimensions: given a trajectory and location for the beginning and end points. Both papers found that the minimum path is comprised of maximum rate turns and straight line segments. This initial work on shortest paths is extended by Chitsaz and LaValle, 2007 to take a third dimension into account: for low altitude ranges the shortest path is the Dubins path in the x-y plane and a constant rate altitude climb. High altitude climbs diverged from the Dubins path due to helical climbing component; however this is not relevant as only low altitude climbs are considered in this project.

McGee et al., 2005, Techy and Woolsey, 2009 apply Dubins shortest path in uniform wind. This is done by considering a ground reference frame and wind reference frame which results in the Dubins minimum path being calculated in the wind reference frame. The maximum rate turns in the air frame of reference "correspond to trochoidal paths in the inertial (ground) frame" [Techy and Woolsey, 2009]. Assuming uniform and time-invariant wind leads to potential inaccuracies which are considered through using 'a turning rate less than the actual maximum turning rate' [McGee et al., 2005]. Both papers considered utilise different approaches to obtain a final optimal path, though the resulting optimal paths are both comprised of a combination of trochoidal path sections and straight line sections. These papers present an extension of the Dubins path concept to consider wind but also outline the added complexity of uniform wind without resulting in a vastly different path.

Savla et al., 2005b, Savla et al., 2005a, Le Ny, 2008 consider the DTSP. The basic approach considered in all papers is in the form of the alternating algorithm which requires calculation of a minimum tour using euclidean distances for an initial order-

ing. From the initial order the heading at nodes is given by the direction of either the vertex before the node or the vertex after the node. With the order, location and heading defined at each point the Dubins shortest path can be calculated. Savla et al., 2005a goes on to consider stochastic DTSP where the points are normally distributed and puts forth a bead tilling algorithm to improve the performance which is an important consideration for the initial planning aspect of this paper. Le Ny, 2008 however goes beyond the scope of this paper in considering variable vehicle dynamics.

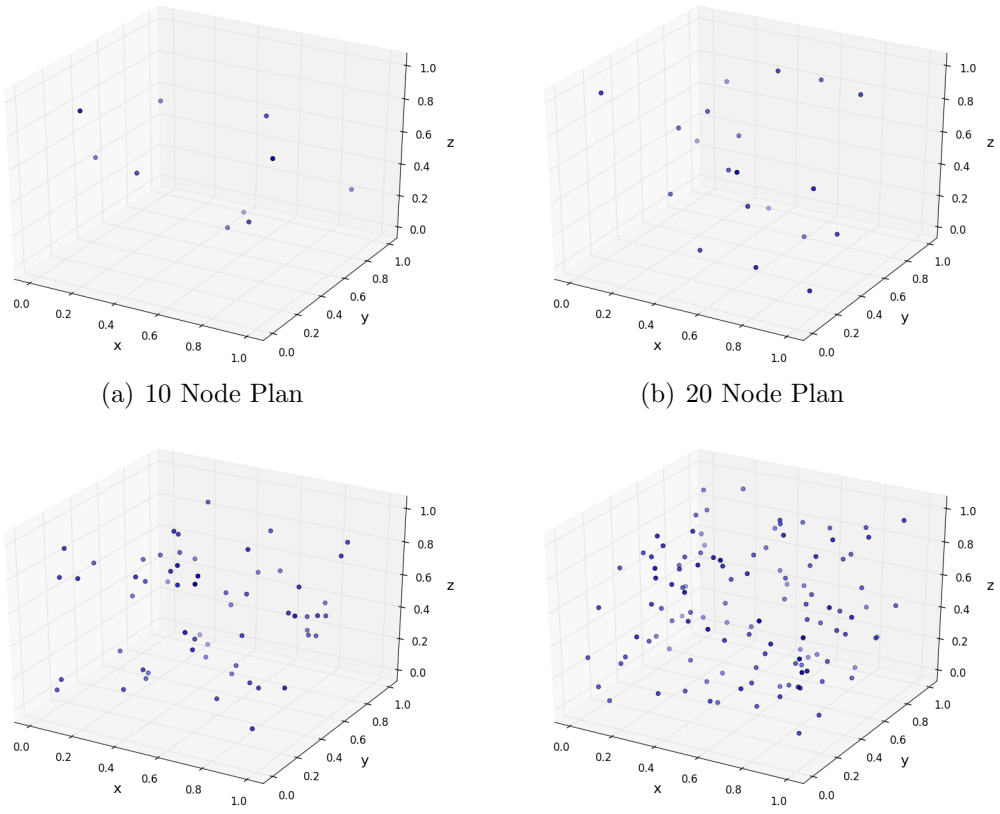
Al-Sabban et al., 0, Chakrabarty and Langelaan, 2009, Langelaan, 2007 look into the minimum energy paths through non-uniform wind vectors by considering the total energy of the UAV and attempting to minimise the reduction in energy. Al-Sabban et al., 0 uses a markov decision process to plan a route through time varying wind vectors which have a degree of uncertainty. Chakrabarty and Langelaan, 2009, Langelaan, 2007 however use a predetermined knowledge of the wind with the aim to exploit atmospheric energies. The given equations of energy and considerations of optimal routes through complex wind fields applies directly to this paper; however attempting to tap into soaring flight is not feasible given the requirement to fully investigate a particular research area.

Forrester et al., 2008, McKay et al., 2000 consider efficient sampling plans for black box experiments to improve the quality of the model produced. Both papers present Latin hypercube sampling to be an improvement on random sampling as they ensure "that each of those components is represented in a fully stratified manner" [McKay et al., 2000] where those components refer to input dimensions. Forrester et al., 2008 extends this by optimising Latin hypercubes to result in the plan with best space fillingness. The sampling plans provided can easily be utilised in the primary planning component of the UAV tour in this project. A. Forrester, 2009 additionally looks at adding samples to existing surrogates to improve the quality of the model produced. These papers provide the basis for optimal sampling plans for both initial data collection and subsequent flights given an existing model. This outlines the importance of a versatile path planner that can cover an area of interest in addition to any collecting of nodes no matter how sparse.

3 Background Concepts

3.1 Latin Hypercubes

Latin hypercubes are sampling plans that provide the best space fillingness while limiting the total number of sampling points required. This is generally applied to testing of computer simulations where the collection of each point is expensive. In this situation however the travel between the points is the expensive component.



(a) 10 Node Plan

(b) 20 Node Plan

(c) 60 Node Plan

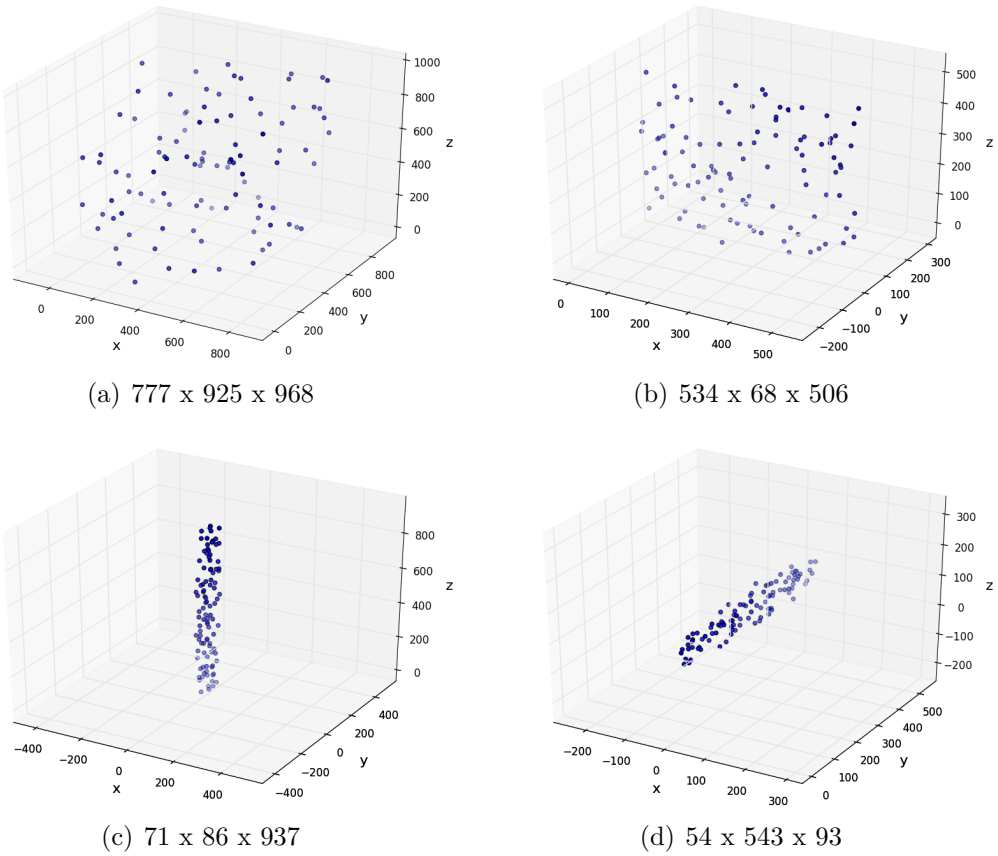
(d) 120 Node Plan

Figure 1: Latin Hypercubes with Varying Numbers of Nodes

Figure 1 shows Latin hypercubes with varying numbers of nodes. The nodes within the Latin hypercubes are located in such a way that ensures along each vertex the nodes are equally spaced and the spread of the nodes is maximised. The resulting plans are therefore efficient sampling plans that are space filling within the area of interest. The Latin hypercubes shown here are computed within MATLAB using code from the work of Forrester et al., 2008 and imported into python for utilisation within this project. Once a Latin hypercube is computed the result is cached so for any following calls MATLAB is not required to be called.

Given that data collection is rarely within a unit cube and more likely on the scale of thousands of meters in a research area that is far from a cuboid, these sample plans need to be altered for use. To apply these Latin hypercubes to provide sampling plans for any research area they can be stretched in each vertex.

Figure 2 shows a number of 100 node Latin hypercubes stretched to varying research areas. It can be seen that the data spread of the Latin hypercubes remains the same along each of the vertexes; however in terms of actual distance between nodes in the sampling plan this causes bunching. This results in data that is equally sampled for each input variable therefore the effect of each variable on the model is equally considered.



(a) 777 x 925 x 968

(b) 534 x 68 x 506

(c) 71 x 86 x 937

(d) 54 x 543 x 93

Figure 2: Stretched 100 Node Latin Hypercubes

3.2 Exact Travelling Salesman

To calculate the least cost tour of a number of nodes the TSP presents itself. The TSP is well documented for many route planning problems, both for two dimensions and three dimensions. The standard form of the TSP is to calculate the least cost of visiting every node where the cost is defined as the euclidean distance between nodes. Due to the euclidean properties of the TSP there are many heuristic approaches to computing best guess solutions.

Without the utilisation of heuristic approaches the TSP is a very computationally expensive problem. This is due to the number of routes that are possible given even a small number of nodes. The number of routes increase by a factor of the total number of nodes each time a node is added to the computation. This is due to that new node needing to be considered at every point in every existing route.

Figure 3 shows an arrangement of nodes with all possible connections shown in blue and the shortest route in green. The number of connections between 6 nodes route is 15. Given that the number of routes through these connections totals to 720 this shows how the TSP can easily become unmanagable.

The optimal route is a found using python. A matrix of the length of all connections is calculated using the euclidean distance, then the route distances are found from

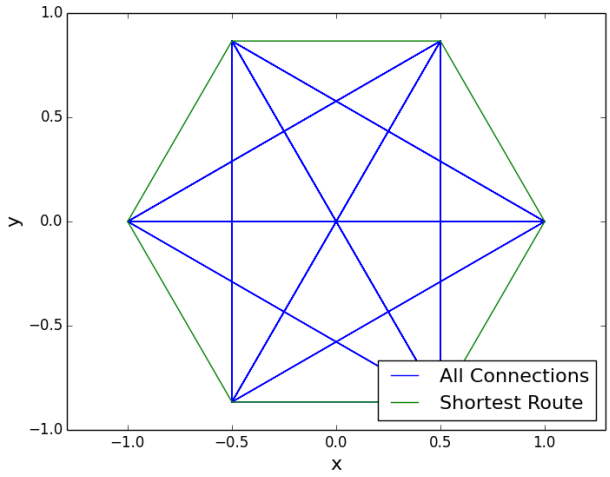


Figure 3: Connections and Shortest Route through 6 Nodes

summing up the individual connection distances from the distance matrix. For all permutations of node ordering, the length of the path is computed and the shortest route is selected.

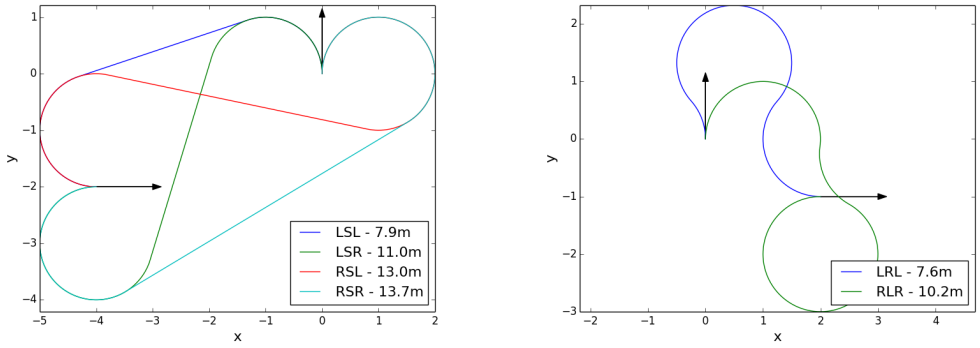
For the route planning in this report it is imperative to devise a TSP solution that provides a best guess method to finding a least cost route. This is because the number of nodes required for a sampling plan to achieve a high quality model will far exceed 10 nodes, resulting in excessive computation time.

3.3 Dubins Paths

The shortest path between two locations when considering purely a start and end location is the euclidean distance between each node. This is the distance that would be experienced by a vehicle that can travel in any direction regardless of current heading. To work out the path length between two points where the start and end directions are determined is a more complicated problem. This requires the considering of the minimum turning radius for the vehicle in question.

A Dubins path is a minimum distance path between a given start position and direction and a given end position and direction. There are a number of different forms of Dubins paths that can be achieved and the minimum path is the minimum of the Dubins paths that can be computed. These paths are comprised of maximum rate turns and straight line segments.

- RSR - Right Turn, Straight Travel then Right Turn
- RSL - Right Turn, Straight Travel then Left Turn
- LSR - Left Turn, Straight Travel then Right Turn
- LSL - Left Turn, Straight Travel then Left Turn
- RLR - Right Turn, Left Turn then Right Turn
- LRL - Left Turn, Right Turn then Left Turn



(a) Turn Straight Turn (b) Turn Turn Turn

Figure 4: Dubins Paths Comprised of Turns and Straight Line Segments

To calculate the different Dubins paths that are possible geometric relations and vector identities can be utilised. The initial stage is to consider the start and end directions. For both the start and end directions the circles that correspond to maximum rate turns are computed. That is the circle where the initial direction is tangential to the circumference and the circles where the final direction is tangential to their circumference.

These paths are calculated in python using the logic detailed by Giese, 2012 to determine the geometric and draw respective paths. The shortest path is then selected by comparing the resulting distances.

For the routes with a straight line segment in the middle, tangent lines are computed. These four sets of lines that are tangential to a start and end circle make up the straight section of the route. The complete path is the combination of an arc that lies in the circumference of the start circle, the straight line section to the end circle and then an arc in the circumference of the end circle. For some cases where the start and end points are close some path types are not achievable.

Figure 4(a) shows the possible routes from the point (0, 0) in direction (0, 1) to the point (-4, -2) in direction (1, 0). The arrows symbolise the start and end headings and the different coloured route symbolise the different routes. In this example the routes comprise of maximum rate turns of radius 1 and straight line segment.

To calculate the paths that are comprised of only maximum rate turns the start and end circles are used again. The circles that lie with their circumference tangential to both a start and end circle is used to determine the points of change from one turn direction to the other. The path then follows the circumference of each of the three circles in turn. Paths of only maximum rate turns are only viable when $D < 4r$ where D symbolises the distance between the start and end points, as above this distance the radius would need to increase to be navigable using only three circles.

Figure 4(b) shows the possible routes from the point (0, 0) in direction (0, 1) to the

point (2, -1) in direction (1, 0). The arrows symbolise the start and end headings and the different coloured route symbolise the different routes. In this example the routes only comprise of maximum rate turns of radius 1.

4 Method

To achieve the objectives of this project a particular process had to be followed. This ensured that each further level of investigation was based on completion of the one preceding it. The initial goal was to design a simple yet effective energy model that could use actual plane data and return the expected energy consumption for a number of flight manoeuvres. Using this energy model as a basis for comparison the computation time of the TSP needed to be improved. Having obtained a least energy route through a sampling plan then create a path through the route that adheres to the UAV flight characteristics.

In this section and for the remainder of this report, **route planning** refers to the ordering of nodes within the sample plan and **path planning** refers to the creating of navigable paths from this ordering of nodes. The route planning aspect requires ranking of comparative energies, however the path planning component requires more detailed energy considerations.

4.1 Energy Model

To correctly estimate the energy used in navigating through a particular route, an energy model to define how different flight manoeuvres consume energy was required. For a basis of the energy model the plane was assumed to consume energy in two ways: in doing work against drag and by doing work against gravity. As the plane navigates the manoeuvres on its route the consumption reduces the available energy in the plane's battery. Given a good energy model the length of the route can be determined a priori safe in the knowledge that the plane will not run out of energy before completing its route.

Obtaining accurate values for the variables required for energy modelling is difficult given an off the shelf UAV. Therefore the energy model was harder to calculate for individual planes. To work around this problem the aerodynamic values for the plane were obtained from results to foil simulations from the internet. This enabled a similar airfoil shape to be selected that allows aerodynamic variables to be easily obtained. The website *airfoiltools.com* contains the results to simple simulations on many airfoils and the data can be pulled for virtually any foil shape at varying Reynolds numbers.

Figure 5 shows the logic behind the approach used to compute an energy model using

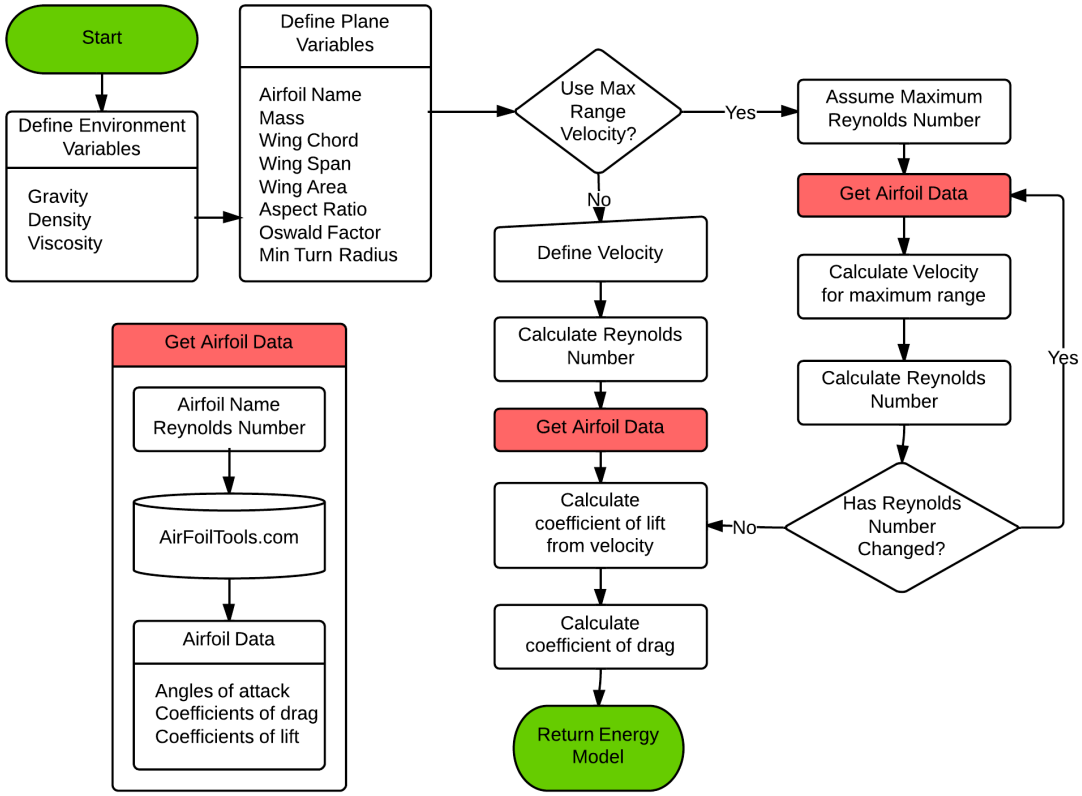


Figure 5: Diagram of Energy Model Logic

readily available atmospheric and plane variables. The best Reynolds number was then selected an iterative approach which was used to download airfoil coefficients. The following equations were used to produce the energy model.

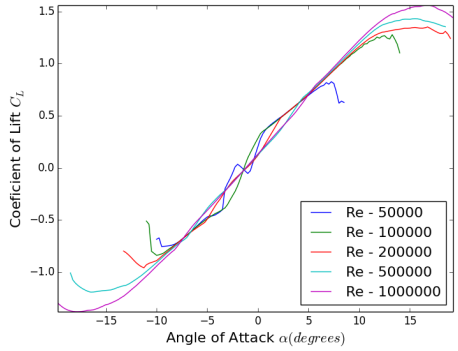
$$Re = \frac{\rho c V}{\mu} \tag{1}$$

Equation 1 was used to calculate the Reynolds number given the flight conditions of the plane. This was required to select the correct data set for obtaining the relevant values for coefficient of lift and the zero lift coefficient of drag.

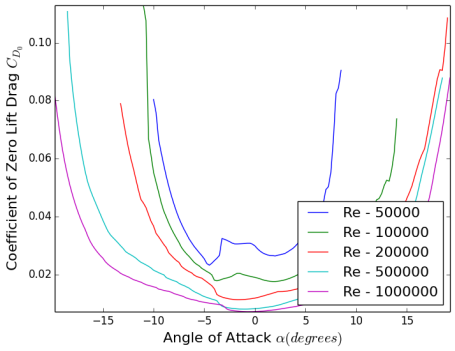
Figure 6 shows the coefficient of drag and coefficient of lift for varying angles of attack. This is data obtained directly from *airfoiltools.com* for the NACA23015-IL airfoil operating in conditions with a Reynolds number between 50000 and 1000000.

$$C_L = \sqrt{C_{D_0} \pi A e} \tag{2}$$

Equation 2 shows the relationship between the coefficient of lift and the zero lift coefficient of drag where the plane's range is a maximum. The data from *airfoiltools.com* was then used to find the values of the coefficients where the equation is satisfied. This calculation was only required if the flight velocity of the plane was not defined.



(a) Coeficient of Lift C_L



(b) Coefficient of Zero Lift Drag C_{D_0}

Figure 6: C_L and C_{D_0} Plots of a NACA23015-IL Airfoil

$$V = \sqrt{\frac{2mg}{\rho S C_L}} \tag{3}$$

Equation 3 depicts the relationship between velocity and the coefficient of lift. This equation was utilised in two ways. When the velocity of the plane is defined this equation defines the coefficient of lift. When however the coefficient of lift is calculated from equation 2 this equation is used to define the velocity.

$$C_D = C_{D_0} + \frac{C_L^2}{\pi A e} \tag{4}$$

Equation 4 was used to amend the coefficient of drag for a 3 dimensional wing. This utilises the Oswald factor and aspect ratio which are parameters used to define the wing loading and shape profile respectively.

$$n = \frac{L}{W} = \sqrt{1 + \left(\frac{V^2}{Rg}\right)^2} \tag{5}$$

Equation 5 shows the relationship between the load factor and the turning radius for the plane. The load factor is the ratio of lift to weight during a turn. This allows the greater energy required in turns to be considered for path planning.

$$E = \frac{1}{2}\rho C_D S V^2 D \tag{6}$$

$$E = n\frac{C_D}{C_L} mgD \tag{7}$$

$$E = mgH \tag{8}$$

Equations 6, 7 and 8 were used to calculate the level, turning and climbing flight consumption of energy respectively. Climbing flight here only considers the work against gravity. Therefore for any form of climbing flight the energy consumption due to drag must also be taken into account. For example if a plane were to fly $1m$ horizontally and $1m$ vertically then the energy used would be found by summing equation 6 with $D = \sqrt{2}$ and equation 8 with $H = 1$.

The energy model was then simplified for the route planning. In the route planning the actual energy does not matter but the ranking of energy consumption does. For this reason the accurate equations for energy are reduced to a form where the energy is merely a cost balance between level and climbing flight. This is similar to the L/D ratio.

$$E = \gamma(\beta D + H) \tag{9}$$

$$\beta = \frac{\rho C_D S V^2}{2mg} \tag{10}$$

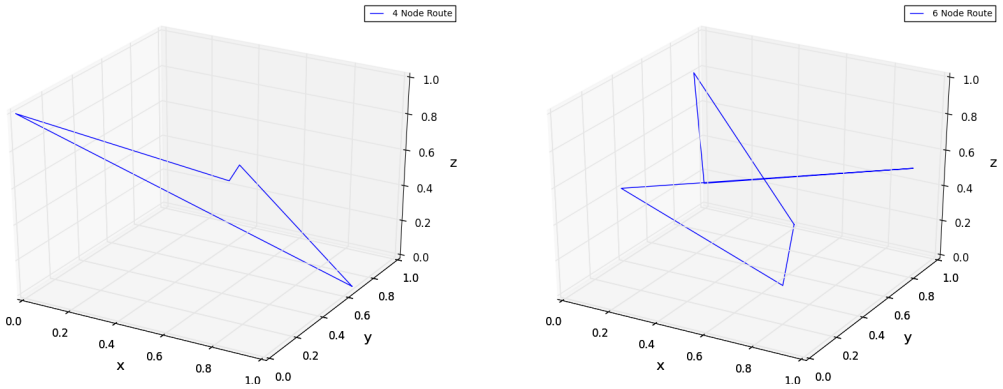
$$\gamma = mg \tag{11}$$

Equation 9 is the overall energy equation used to calculate accurate energy based on distance and height gain. The constants β and γ are depicted in equations 10 and 11 respectively. The energy model has been manipulated in this way to illustrate that the γ component is merely a factor applied to the whole equation. Therefore for the purpose of routing the constant α can be used alone to define the energy equation. This abstraction provides for the constant γ to not affect route selection. Therefore it can be applied subsequently to calculating the ordering of nodes.

From this point forward in the report the energy computation will be referred to in two ways: the **cost** of a route will refer to the component of equation 9 contained within the brackets and is an adjusted length measured in meters, the **energy** however refers to the result of the full equation and is measured in joules.

4.2 Exact Travelling Plane

The exact travelling plane developes on the TSP to include energy considerations for route planning. Firstly the cost of travel between every possible pair of nodes in the sample plan was calculated. Then for every permutation of how these nodes can be ordered the cost of the overall route was calculated and the least cost selected as the optimal route.



(a) 4 Node Route (b) 6 Node Route

(c) 8 Node Route (d) 10 Node Route

Figure 7: Exact Travelling Salesman Routes

Figure 7 shows the optimal routes for different numbers of nodes. These optimal routes are found by computing the exact cost of each and every route option. Although this yields the shortest route, this approach was not efficient in terms of the computation time required. To enable calculation of a 10 node route the start point of the route was defined by the lowest node. This reduces the complexity of the path planning problem to that of a 9 node route.

Number of points	4	6	8	10
Number of possible routes	6	120	5040	362880
Computation time (ms)	0.0	1.0	43.0	3658.0
Best route cost (m)	1.16	1.11	1.11	1.14

Table 1: Comparison of route calculation

Table 1 shows the number of possible routes and the resulting computation time given different numbers of nodes. For a standard travelling salesman problem the number of possible routes is defined by $n!$ however in this case the number of route options is equivalent to $(n-1)!$ where n is the number of nodes in each case. This is due to the start node being defined, thus the complexity is reduced by a single node. The number of routes directly relates to the computation time.

The computation time of the exact TSP far exceeds what would be practical for this project; therefore the performance has to be increased to produce workable routes

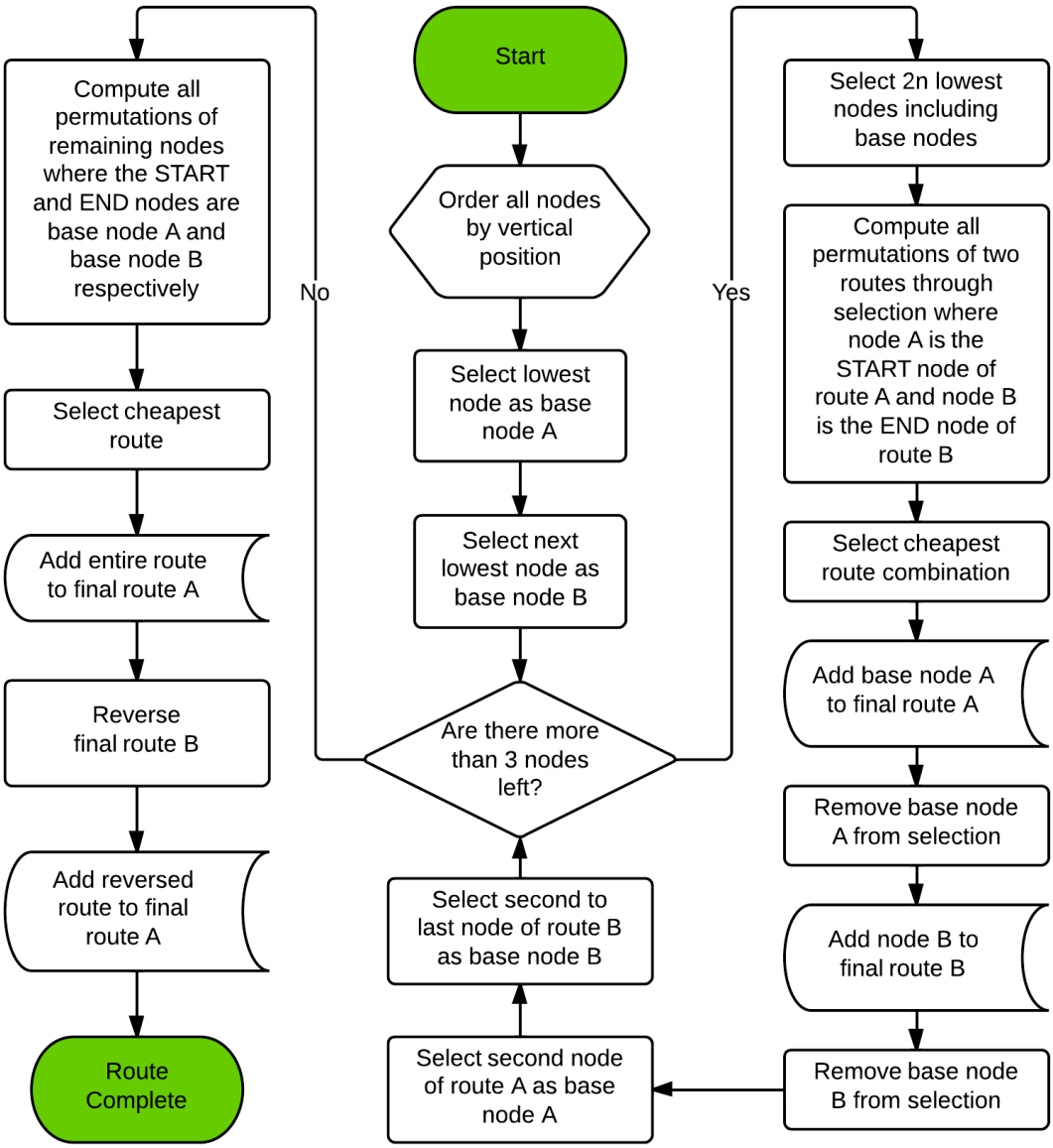
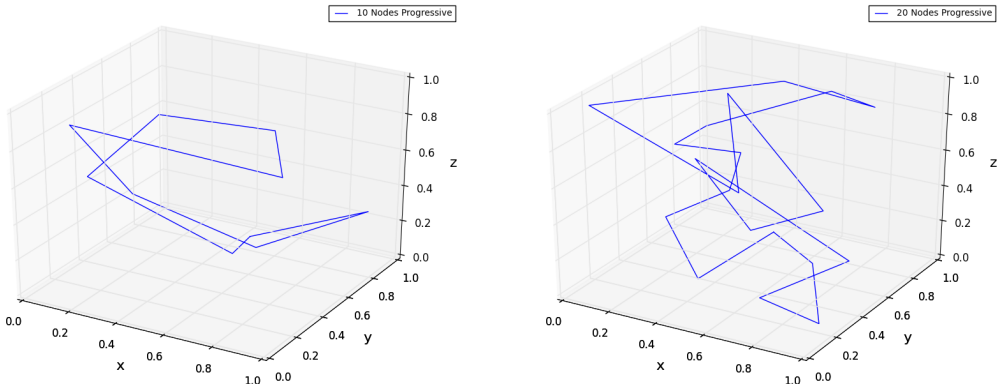


Figure 8: Progressive Travelling Plane Logic

from the number of nodes required. The heuristic approach used in this report is taken from consideration that the best routes in figure 7 are generally comprised of a single climbing component and single descending component.

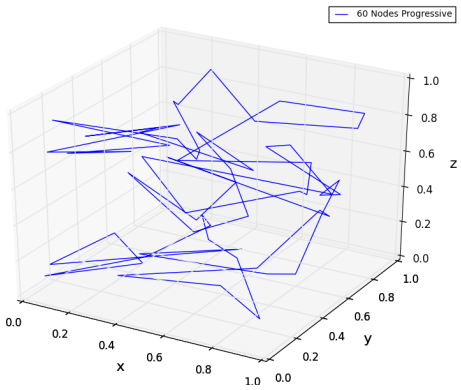
4.3 Progressive Travelling Plane

From the observation that the optimal routes from figure 7 all comprised of an up component and down component, the logic for a best guess approach was devised with far less computation time required. It seemed very logical that a route through nodes in three dimensions would comprise of an up section and down section however it was important to consider the absolute optimal solution.

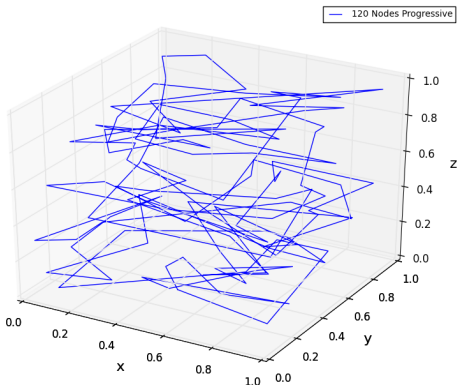


(a) 10 Nodes Progressive

(b) 20 Nodes Progressive



(c) 60 Nodes Progressive



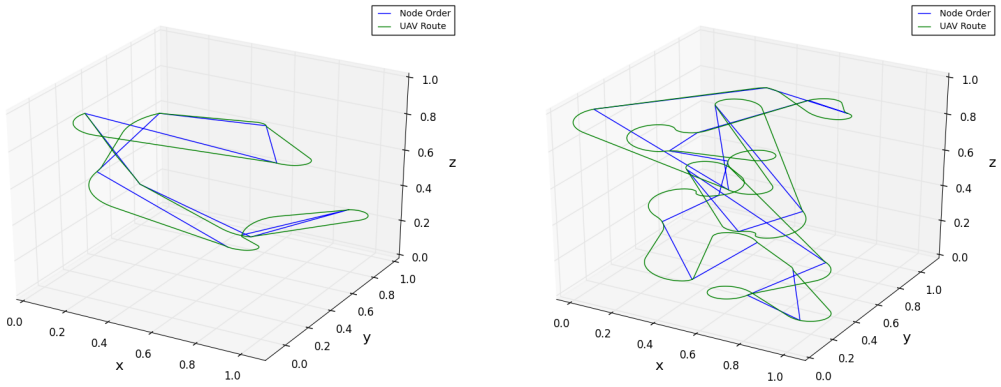
(d) 120 Nodes Progressive

Figure 9: Exact routes calculated by travelling salesman

Figure 8 shows the logic behind the progressive travelling salesman function. The function works using an iterative approach to selecting the optimal route. A small subset of the lowest nodes is analysed at each stage to define the route. Here n defines the number of nodes in each route within the subset. This iterative approach assumes that the least cost route will be comprised of an up component of travel and down component of travel. Upon path planning for each level of the sample area a single point is added to each route. The up route is constructed from beginning to end while the down route is constructed in reverse. When the routes meet at the top, route A and route B are joined to form a single route.

Figure 9 shows a number of optimal routes for varying numbers of nodes whose order is defined by the progressive travelling salesman algorithm. For the 120 node route it is difficult to see the exact routing; however for the other routes a logical approach to the routing problem can be seen. The initial progressive travelling plane logic did not produce routes of this quality as the least cost route on the way up would always favour finishing lower down even though in the next iteration this would mean having to gain greater height. Due to this visual identification of problems the cost of travel to the highest node is added to the up path for each iteration. This improvement caused the results that are shown.

Figure 9(b) displays some imperfections in the calculation of an optimal route. This



(a) 10 Node Path (b) 20 Node Path

Figure 10: UAV Routes and Paths through Latin Hypercubes

is due to the significant change in height experienced at the top section of the route (where the routes A and B join up to form a single route). This suggests that figures 9(c) and 9(d) could contain inefficiencies in the route.

4.4 Path Planning

The least energy route through a number of nodes has been defined; however this route assumes that the UAV is able to turn on the spot and is not constricted by turning radius. Therefore to compute the actual energy cost of circumnavigating a route the turning radius of the UAV needs to be considered. Dubins paths can be used to produce a path from this route.

To compute a path through a sample volume the altitude of the unmanned aerial vehicle must be taken into consideration. The assumption for this stage in path planning is that the plane considered is able to change its rate of climb quick enough to approximate being instantaneous, whereas the rate of turn is not able to change with the same speed.

To increase the accuracy of this assumption the rate of climb for each section would change gradually from the previous section's rate of climb to the next section's rate of climb. However given the added complexity of this implementation it was assumed that the improvement on accuracy would not be sufficient to warrant the time required.

Figure 10(a) shows the optimal path and route for a UAV to circumnavigate a 10 node Latin hypercube. The route is calculated before the path and then the path is calculated from the heading at each node in the route. From visual observation of the route it looks to be an optimal ordering of nodes and the path selected through the nodes adheres to the flight characteristics of a plane.

Figure 10(b) shows the optimal path and route for a UAV to circumnavigate a 10 node Latin hypercube calculated in the same manner as the previous. Though this path is

harder to visually inspect it looks to follow a path with limited change in height. In addition for each change in direction the flight characteristics are taken into account.

5 Results

5.1 Energy Model Results

In section 4.1 an energy model for the flight of a plane was defined. The varying coefficients of lift and drag were investigated to validate the iterative approach to define the Reynolds number. An example plane was selected to test whether the results for this model are viable. The energy coefficients of the plane considered defines the values taken forward in the report.

Air Foil Name	naca23015-il
Mass (kg)	$m = 0.64$
Oswald Factor	$e = 0.7$
Wing Area (m^2)	$S = 0.204$
Wing Span (m)	$b = 1.50$

Table 2: Table of Plane Properties

Table 2 shows the properties of the plane considered. Aside from the name of the airfoil and the Oswald factor, all the parameters are readily available specifications that can be found on many out of the box UAVs. The airfoil was to be specified in order to obtain an estimation for the coefficients of lift and drag. The Oswald factor displayed in this table is also an estimation for the given plane; however for the purpose of this analysis is probably sufficient.

Aspect Ratio	11.03
Wing Chord (m)	0.14

Table 3: Table of Calulated Plane Properties

Table 3 shows a number of calculated properties from the initial data. The aspect ratio was calculated using $A = \frac{b^2}{S}$ and the wing chord was calculated using $c = \frac{S}{b}$. This allowes for any plane whose data can be fitted to the data in table 2 to be used for this energy model.

Given the main driving variable of analysis is the energy coefficient β, the energy coefficient output from the energy model was investigated. The flight velocity defines how great the drag force is that the plane works against in normal flight; therefore this was used as the driving variable to analyse the changing energy coefficient.

Figure 11 shows the variation of the energy coefficient with flight velocity. The separate series define when the Reynolds number used to obtain the foil data has changed

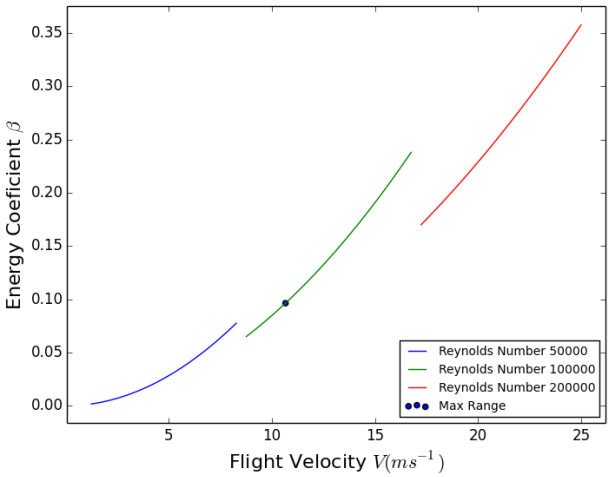


Figure 11: Flight Velocity V and Energy Coeficient β

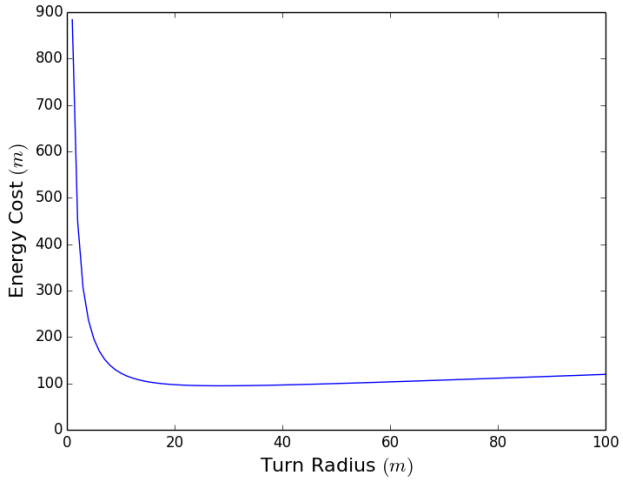


Figure 12: Energy Cost of 180° Turn at $10.7ms^{-1}$

due to the velocity change. The Reynolds number can be plotted as a separate series as it is returned when the velocity of the plane is altered. For actual flight this curve would be a single line; however as the data sets are only available for certain Reynolds numbers the inconsistencies are present.

The velocity that yields the greatest range given the plane considered is $V = 10.7ms^{-1}$ at this flight velocity the energy coefficient takes the value $\beta = 0.10$. For investigation of climbing and turning flight this is the flight velocity considered.

Figure 12 shows the variation in the cost of the plane detailed in table 2 navigating a 180° turn with varying turning radiuses. The varied turning radiuses cause the distance travelled to vary hugely. Thus if this analysis was only done with the circular distance of the turn, the cost would mainly be affected by this distance travelled. Therefore for this analysis the plane is assumed to start at one point, then, travel in a straight line until initiating a turn which passes through a second point half way round

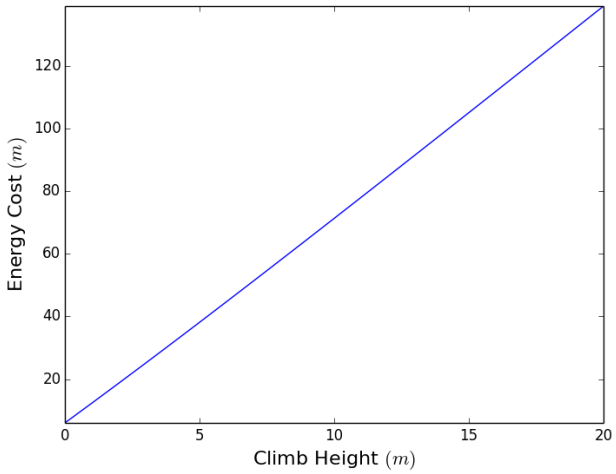


Figure 13: Energy Cost of Flight with Horizontal Distance $10m$ and Varyed Vertical Distance at $10.7ms^{-1}$

the turn before finishing the turn and flying back to be level with the initial point. This means as the turning radius is increased the straight line flight is decreased.

It can be seen from figure 12 that the energy consumed in turning 180° reduces to an optimal at a given turning radius. This is due to the lesser forces required on the plane to turn with a greater turning circle. This finding needs to be considered when it comes to path planning.

Figure 13 shows the variation of the cost for the plane detailed in table 2 to climb for a horizontal distance of $10m$ over varied climb heights. Here the horizontal distance is maintained at a constant and the vertical distance is increased to determine the extra flight cost. It can be seen that cost varies fairly linearly with climb height. This relationship is not truly accurate as it does not account for the reduction in aerodynamic efficiency which results from a greater angle of attack that is required for a greater climb angle.

The results to the energy model are as expected given the simplicity of the model utilised; however there is room for improvement on the accuracy of how the model predicts climbing and turning flight consumption. The framework for allowing any angle of attack to be pulled from the internet makes this improvement easy to facilitate.

5.2 Progressive Travelling Plane

In section 4.3 a best guess approach to the TSP was presented with specific application to the flight of a plane. This provided a method to calculate the order in which nodes should be visited for the lease cost route. The method was sufficiently computationally simple so that routes could be found through large numbers of nodes.

To test the progressive algorithm fully the input parameters that affect its operation were varied to consider how they affected the resulting best guess cost. The following parameters affect the operation of the progressive travelling plane algorithm:

- Number of nodes sample plan N
- Number of nodes in subset n
- Energy coefficient β

Before considering the quality of the model produced the parameters affecting the computation time were investigated. This allowed the further investigation of parameters to stay within reasonable limits. The energy coefficient does not change the computation time as the problem is the same complexity but it drastically changes the optimal ordering.

Nodes in sample N	10	10	10	10
Nodes in subset n	2	3	4	5
Computation time (ms)	2	3	52	4083
Best route cost (m)	1.27	1.26	1.22	1.22

Table 4: Comparison of route calculation

Table 4 shows the effect of increasing the number of nodes considered in the subset, where subset refers to the number of nodes that are considered in each route of the progressive iteration of two routes. It is apparent that the computation time increases dramatically as seen in the exact TSP as the number of nodes in the subset is increased. This means computationally this method is not viable beyond 4 nodes in each route in the subset.

Nodes in sample N	10	20	60	120
Nodes in subset n	4	4	4	4
Computation time (ms)	52	179	725	1626
Best route cost (m)	1.22	1.4	2.47	4.52

Table 5: Comparison of route calculation

Table 5 shows the effect of increasing the total number of nodes in the routing problem while maintaining the number of nodes in the subset at a constant. It can be seen that for 120 nodes in the subset the computation time remains manageable as it did not increase drastically with greater numbers of nodes in the total sample.

The computation requirement of the progressive TSP is more than acceptable. However the routes produced may not be sufficiently close the actual optimum. Visual inspection of the given routes suggested that the logic is sound however the cost needs to be compared with the cost of the absolute optimal route. This is done by computing the costs of all routes using the exact TSP and then comparing.

Figure 14 shows a histogram of different route costs for a 10 node Latin hypercube. The lines on this histogram plot represent the best cost routes with different numbers

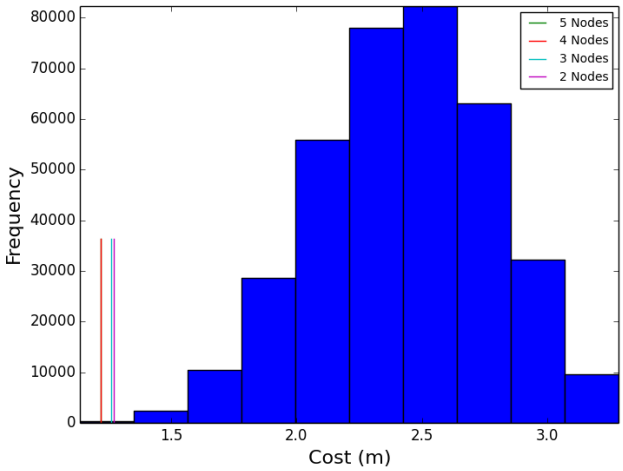


Figure 14: Histogram of 10 node route costs

of nodes in the subset. It can be seen that the cost of the best route from the progressive approach closely approaches the optimal solution as calculated using the exact TSP.

The minimum cost route for the exact travelling plane was $1.14m$ while the minimum cost for the progressive travelling plane was $1.22m$; a figure is within 7%. In terms of ranking the best guess result ranks 28 out of 362880 results. Which means the progressive best cost route is within 0.01% of the rankings of all possible routes. Therefore for a 10 node route with the energy coefficient $\beta = 0.1$ the progressive travelling plane approach is very much acceptable.

To fully test validity the progressive travelling plane route planner should have been compared with the exact results for Latin hypercubes with a number of nodes greater than 10; however the computation costs of a more than 10 node routes makes this not a viable option. As an alternative, the relative cost decrease of adding more nodes can be considered.

Figure 15 shows how the relative route cost varies with the number of nodes in the subset. Relative route cost refers to the cost as a percentage of the minimum cost achieved. It has to be noted at this point that the minimum cost achieved is not the actual minimum cost route. However it does mean the relation of the other costs can be seen to the best computable cost and allows for comparison between vastly different numbers of nodes. From looking at this figure it can be seen that routes with less total nodes generally plateau within the range of the subset numbers investigated. This shows that the result achieved approaches some form of minimum. However, given the computational time to compute a route with 6 node subsets is vast this is the furthest this analysis can be taken.

For the investigated energy coeficient $\beta = 0.1$ at low nodes the progressive TSP yields

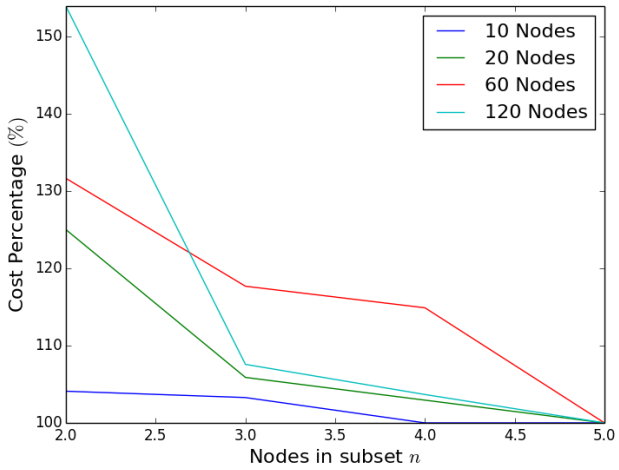


Figure 15: Comparison of Relative Route Costs for Progressive Travelling Plane

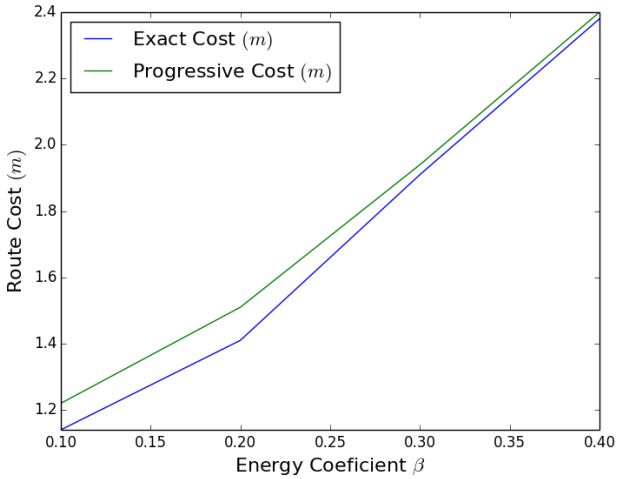


Figure 16: Best Costs of Routes through 10 Node Latin Hypercubes with Varying Energy Coeficients

good results. However, given the energy coefficient β depicts the relative cost of level vs inclined flight and that the progressive travelling planes is based on route planning through height order; changing this coefficient could have a big effect on the quality of the progressive approach.

Figure 16 shows how the progressive and exact travelling plane results are affected by changing the energy coefficient β. From looking at the results in this figure it can be seen that the progressive travelling plane approach holds true even when the energy coefficient is varied.

The progressive travelling plane approach to route planning has been validated under varying parameters with links to the exact travelling plane model. An attempt has been made to verify that the quality of the results hold true for larger numbers of nodes however due to the inability to test this hypothesis directly this is not verified.

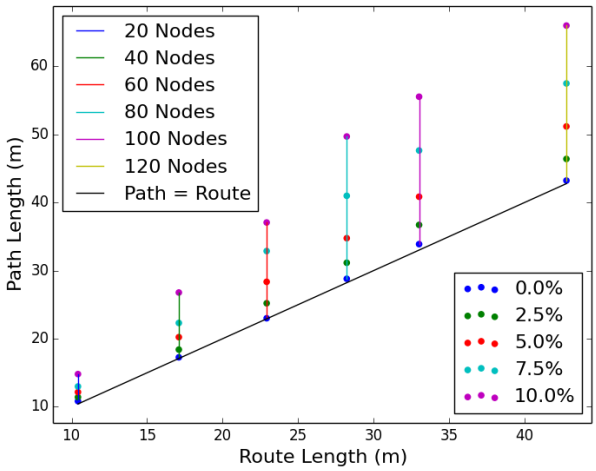


Figure 17: Comparison of Path and Route Length for Varyed Nodes and Turning Radiuses

The computational efficiency of the progressive TSP has been found to be far better than that of the exact TSP. As a result of these factors and given the extent of this analysis, the progressive TSP is acceptable to take forward to the next stage of the project.

5.3 Path Planning

In section 4.4 Dubins paths were utilised to produce navigable routes from a set of ordered nodes. The paths produced passed visual inspection of validity. The resulting effect on both path length and energy consumption are investigated to consider the importance of path planning for UAVs.

Figure 17 shows the relationship between the route length and path length for a number of different Latin hypercubes. For each Latin hypercube the distance of the shortest route (length of ordered route through nodes) and a number of shortest paths (length of path through ordered nodes that takes into account the flight characteristics of the plane) have been calculated and compared. The shortest paths are considered with turning radiuses varied between 0% and 10% of the length of the side of the area that is being explored. For this analysis the area of interest is a unit cube and the percentage value represents the maximum turning radius of the plane over the length of one axis.

It can be seen from figure 17 that as the number of nodes in the Latin hypercube is increased (vertical coloured lines indicate a set of tests on a single Latin hypercube) the effect of increased turning radius (turning radius is indicated by sets of coloured points) also increases. For the case where the turning radius is 0% the path length and route length are the same as the UAV can affectively turn on the spot. Given the

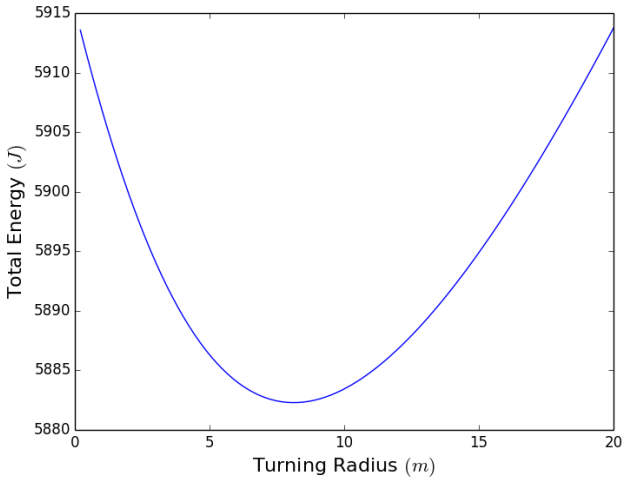


Figure 18: Total Path Energy for Route through 20 Node Latin Hypercube with Changing Turning Radius

size of the research area that will be utilised for collecting atmospheric data using a UAV the added distance due to path length will have a negligible effect on the final distance. The stage at which this consideration may be required to be considered is in computing the final energy of the path. This is due to the greater accuracy of energy cost being required at this stage.

To utilise the path planning component in computing a best estimation of the energy consumed along the path, the energy as a result of turning radius needs to be considered along with that of level and turning flight. The energy consumed in a turn is a result of the severity of the turn and the distance travelled in that turn. The energy model presented in section 4.1 was able to compute the energy cost of navigating a turn at a given radius. The energy consumed was found to have an optimal value where the balance between high force turning angle and long distance turning against drag were at a minimum. For the path planning problem the effect of the turning radius must be considered.

Figure 18 shows the variation of path energy through a 20 node Latin hypercube at 10m/s as a result of altered turning radius. It can be seen that the turning radius affects the total energy consumed on the route and has an optimal value. This optimal value is a product of the spacing of the nodes in the sample plan and the velocity of the plane. The variation in energy as a percentage of the minimum energy experienced is only 0.53% therefore the variation of turning radius on the energy cost is sufficiently negligible to disregard.

Figure 19 shows the variation of total path energy as a result of varied turning radiuses and number of nodes in the sample plan. This figure is to illustrate the effect of node spacing on the energy consumption of a path. The Latin hypercubes with greater numbers of nodes have closer packed nodes therefore the lower turning radiuses are

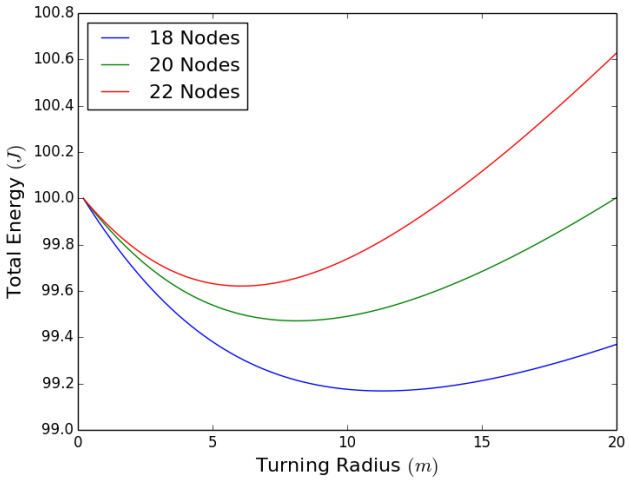


Figure 19: Total Path Energy for Routes through a Number of Node Latin Hypercubes with Changing Turning Radius

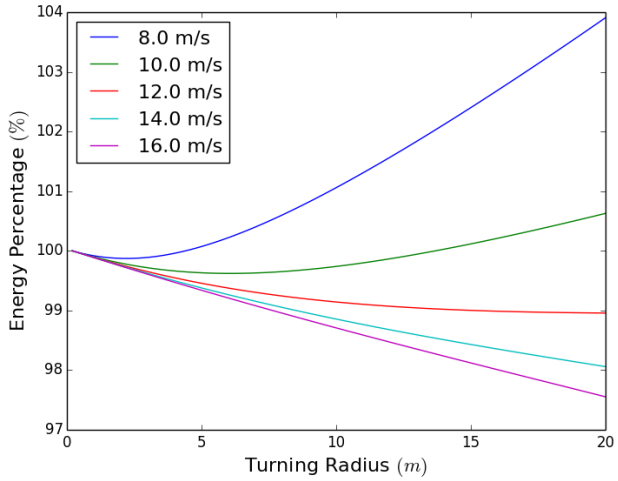


Figure 20: Percentage Path Energy for Route through 22 Node Latin Hypercube with Changing Turning Radius and Flight Velocity

more optimal. The percentage change to the energy of the route is negligible as a result of increase turning radius however this is purely illustrative of the effect of node spacing. The energy percentage here is the energy as a percentage value of the energy required for a route with a turning radius of unity.

Figure 20 shows the variation of total path energy as a result of varied turning radiuses and flight velocities. This figure is to illustrate the effect of flight velocity on the energy consumption of a path. The energy percentage here is the energy as a percentage value of the energy required for a route with a turning radius of unity. It can be seen that the greater the flight velocity the higher the optimal turning radius is. This figure displays percentage values as the energy increases drastically given greater flight velocities.

The path planing component of this report allows the energy consumed by a UAV in navigating a sample plan to be more accurately calculated as the effect of the

turning radius can be investigated. This means the path to be optimised for the best turning radius given a route. Additionally the findings outline the potential for the turning radius to be optimised for least energy consumed based on a node by node basis, this suggestion is due to the optimal turning radius being dependent on node spacing therefore should be calculated per node as the distance between each node is very different. For the continuation of this project the added energy consumption due to the UAV flight path will be disregarded as the energy consumption is not vastly different to that of the route for large turning radiuses in comparison to the research area.

5.4 Sample Plan Model

Thus far in the report routes have been found from a collection of nodes in particular sampling plans, the number of nodes in the sampling plan along with the area of interest affects the length of the route through the points and the total height change experienced. These two factors then go to calculating the total energy expenditure of the route. The desired design route is to calculate the desired route given the area of interest and total energy of the plane. This requires the relationship between the length, width and height of the research area and the number of nodes in the sample plan to the resulting energy cost of the route to be modelled.

The problem with computing this relationship is that the route through the sample area is dependent upon the energy model for a given plane. As in the extremes the energy model can completely favour either climbing flight or level flight.

To compute a model for each Latin hypercube a sampling plan was defined with 100 test cases this sampling plan ranged between 0 and 1000 in each vertex. These sampling plans provide for the most efficient way to collect data on each Latin hypercube. For each Latin hypercube with between 20 and 120 nodes all test cases are analysed to return the exact energy cost. This energy cost can then be related to the research volume for each set of node numbers.

Figure 21 shows the logic required to plan a route based on the total energy available. In this diagram the calculation of model variables is shown in the process. However if these variables were required to be calculated on the fly this would not be a viable approach. In addition if this were to be calculated each and every time the exact research area would be used for each number of nodes, as opposed to 100 samples which are used to return a model for the varying route cost given a number of nodes.

Figure 22 shows the relationship between the research area ($ximesy$), research height (z) and the resulting route cost. This depicts that the energy required to circumnavigate a 100 node Latin hypercube, varies non linearly with both the area and height.

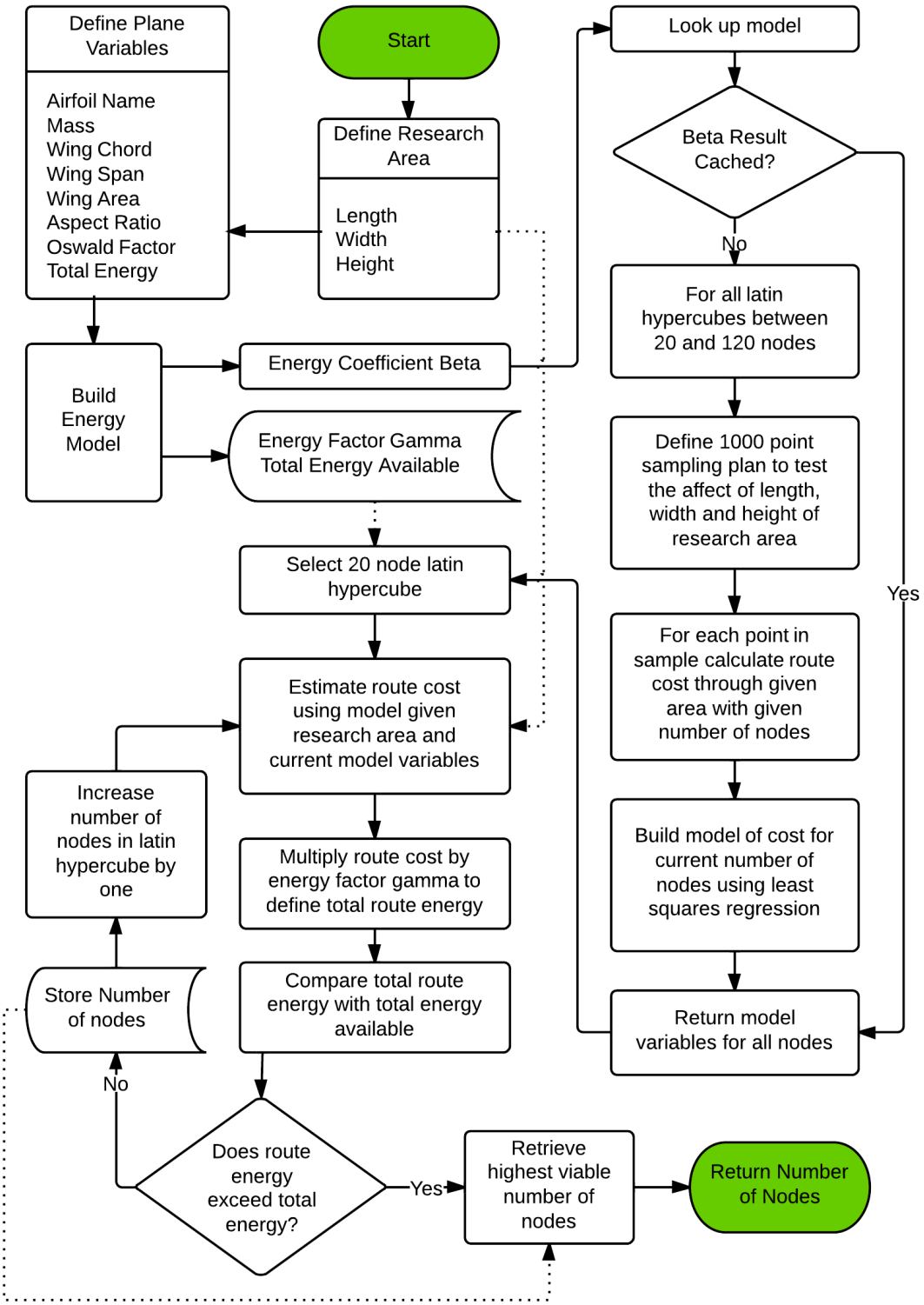


Figure 21: Diagram of Sample Plan Model Logic

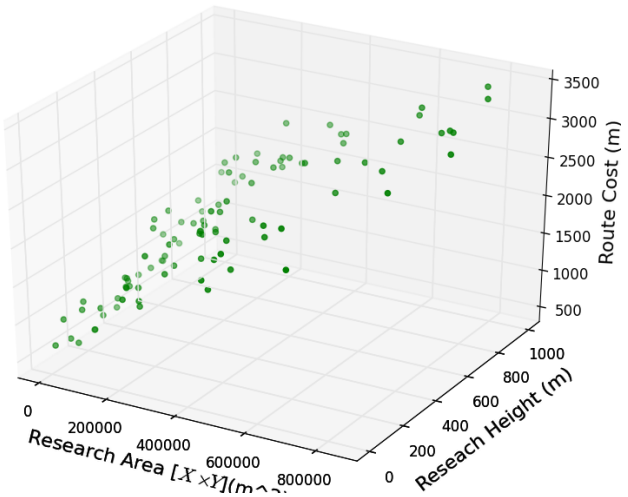


Figure 22: Scatter plot of varying route and path lengths for a 100 node Latin hypercube model

This figure shows that the model needs to be computed from both the length and width as opposed to the area however this form allows for depiction on a figure.

$$E = ma \cdot (xy)^{ea} + md \cdot abs(x - y) + mz \cdot z^{ez} + c \tag{12}$$

Equation 12 shows the format of how the length, width and height of the research area affect the route cost. For every number of node, Latin hypercube and energy coefficients different model parameters are required to correctly depict the relation between the dimension inputs and the energy cost.

$$E = 0.15(xy)^{0.708} + 0.75abs(x - y) + 0.50z^{1.064} + 282 \tag{13}$$

Equation 13 shows how the length, width and height of the research area affects the route energy for a route determined by the energy coefficient 0.1 through a 100 node Latin hypercube. The parameters here are calculated using a least squares regression function.

To calcualte the actual route energy from this cost the energy factor gamma is required. The route cost can simply be multiplied by gamma for the plane considered for routing and this model produces the total route cost for a particular plane without having to compute all costs of routing round different Latin hypercubes.

Figure 23 shows the comparison of the desired route energy and the resulting route energy for the energy coefficient $\beta = 0.1$. The inclined line through the data shows the optimal result where the prediction is completely accurate. This data is obtained from randomly varying the research area and the desired energy consumption and then computing the number of nodes that are predicted to make up the route that

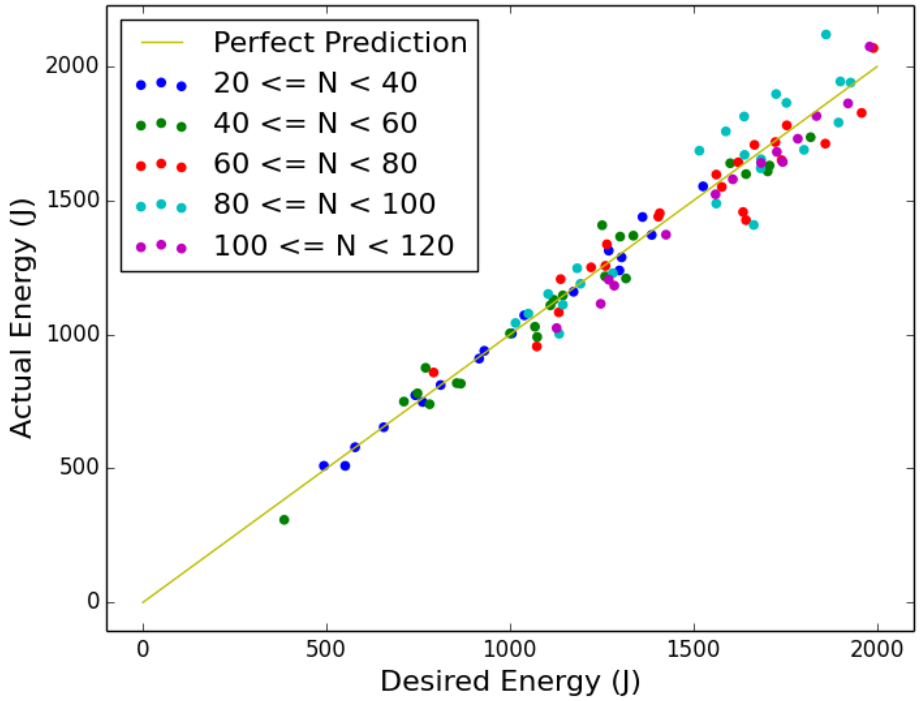
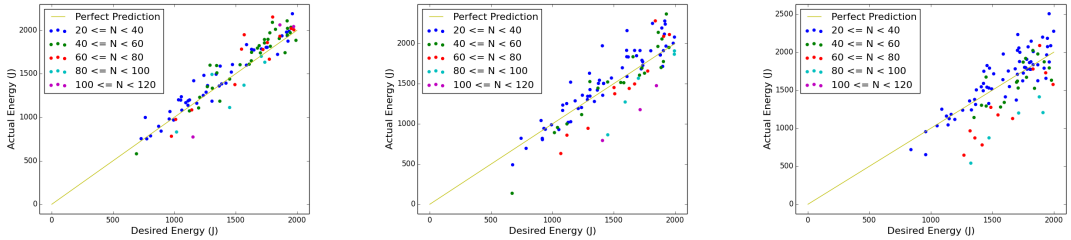


Figure 23: Energy Model Prediction Compared with Calculated Energies For Energy Coeficient $\beta = 0.1$



(a) $\beta = 0.2$ (b) $\beta = 0.3$ (c) $\beta = 0.4$

Figure 24: Energy Model Prediction Compared with Calculated Energies For Varying Energy Coeficients β

is optimal. The actual energy of this route is then calculated exactly and plotted in comparison to the desired route. The series represent the number of nodes used to produce the routes.

The predictions seen for the case where $\beta = 0.1$ are on average out by 4.72% of the desired route energy. This suggests that for this energy coefficient the model produced is viable to accurately select a sampling plan given a required energy consumption. This may not be the case for other energy coefficients due to the drastic affect the energy coefficient has on the routing problem.

Figure 24 shows the comparison of desired and actual route energies for random sample areas and desired route energies for the energy coefficients $\beta = 0.2$, 0.3 and 0.4. It can be seen that as the energy coefficient is increased the prediction accuracy

for the route becomes less viable. Given these predictions are for specific values of the energy coefficient these results would get less accurate given any value for the energy coefficient being utilised. This would be the case for real implementation as the value for the energy coefficient is precisely defined from the plane energy model.

Figure	Energy Coeficient (β)	Prediction Error
23	0.1	4.72%
24(a)	0.2	7.95%
24(b)	0.3	11.87%
24(c)	0.4	12.62%

Table 6: Table of Results

Table 6 shows the prediction error variation as a result of varying the value of the energy coefficient. It can be seen that the model predictions are only viable for low values of the energy coefficient. Therefore the model is of an acceptable accuracy to be utilised in the model predictions for low values of the energy coefficient. Given the UAV considered in this report has a low energy coefficient at optimal range velocity this suggests that this model would be acceptable for use within path planning for the purpose of collecting atmospheric data using UAVs, as this enables a research area to be specified and a suitable Latin hypercube to be selected given the desired total energy expenditure.

6 Conclusions

This report has looked at collection of atmospheric data using UAVs with specific focus on collecting the best set of sample data for a given research area while maximising utlisation of the energy contained within the plane. To complete this task Latin hypercubes were used to provide optimal space filling sampling plans and then the least cost route, with the path then calculated through the Latin hypercubes. To enable planning from the basis of the required energy consumption, the plane's energy consumption was modelled in a mannar where a single coefficient defines the route characteristics. This enabled a model to be produced from multiple tests that can predict the energy consumed based on the number of nodes in the Latin hypercube. This provides for the best spread of data collection and adheres to the UAV energy requirement.

6.1 Energy Model

The energy model produced uses the most simplified aerodynamic equations to determine the cost of navigating a certain path. Given the simplicity of the energy model, routing can be characterised based on a single variable which can be used in

a sample model that determines the optimal number of nodes. The coefficients of lift and drag selected for the plane do not accurately portray the real values that will be experienced as they are pulled for purely the airfoil in question. For the routing component of this project the ranking of energy cost is taken to be more important than actual cost this is acceptable. However the drag effects of the plane fuselage need to be accounted for otherwise a route calculated would far exceed the predicted energy. The values utilised for the air density and viscosity in this model assume that they remain constant at all altitudes in the research area which is not an accurate assumption when altitude varied greatly.

6.2 Travelling Plane

The progressive approach to the TSP yields a good estimation as to the least cost path to navigate a number of nodes. This enables route planning with numbers of nodes that would not be possible using the exact travelling salesman. Therefore the calculation produced is of significant value. The limitation lies in how many nodes the planning can be relied upon to route though while maintaining a best guess cost. Due to the inability to collect data on the perfect solution for more than 10 nodes I have found no sound method for substantiating routing though a 120 node hypercube. The quality of results obtained in the tests performed suggested this route logic to be solid for what has been tested though. This approach provides what it set out to accomplish which is a best guess solution to route planning that is sufficient to be confident of this routing algorithm for the basis of the subsequent calculations

6.3 Path Planning

Utilising Dubins paths to take into account the flight characteristics of the plane enables the energy model to be applied more accurately and the turning radius to be optimised according to the increased consumption related to the bank angle of the turn. Dubins paths are well documented solutions to the problem of vehicle paths and there has been previous literature on applying them in three dimensions. Therefore the shortest distance paths calculated in this report are likely to be correct as any issues can be visually defined. However in light of analysis the shortest Dubins paths do not always correspond to the least energy path due to the increased energy required for a smaller turning radius. Working to improve the implementation of these paths is important as this project strives to compute an as accurate as possible energy cost.

6.4 Data Model

The data model calculated works well for low values of the energy consumption coefficient that directly correspond to the energy coefficient used to produce the model. However for higher values of the coefficient or in-between two tests where there is a lack of data from previous tests the model is not accurate enough for utilisation. With enough test cases where the energy coefficient is varied using a four dimension sampling plan not just the three dimensions which define the area of interest, the model produced is likely to hold sufficient worth to confidently predict costs for any low energy coefficient, not just coefficients that lie in previously tested conditions. The correlations of the sample model are only viable in the confines of the current energy model therefore an improvement on the energy model would require updating the sample model to reflect the added complexity.

6.5 Looking Forward

The approaches documented here are far from a perfect approach to the problem of collecting atmospheric data. However the combination of sampling plans and the progressive travelling plane approach to path planning for atmospheric data collection presents an interesting foothold for further investigation. In addition the proof that, in its simplest form a model can be produced that allows planning from the basis of the required energy consumption, facilitates the potential (however small) for this work to be utilised in actual path planning for atmospheric data collection.

An initial aim of the project was to include consideration for uniform wind which would make the path planning presented here closer to the optimal. Dubins paths in the wind frame of reference correspond to trichordial path segments in the ground frame of reference. This would require a further geometrical calculation to enable this consideration to the path planing. In terms of the routing problem a previous energy model for this project included the consideration on computing the line integral through a vector field that allowed the distance of travel to be altered for the prevailing wind. This means that the cost of travel from nodes could be altered to consider even a non-uniform wind, however was not included for the final report due to the complexities of implementation.

Given the energy model is the basis for all routing consideration in this project, a non-linear approach to energy modelling would have the facility to correctly ascertain the cost of not just changing height but consider the rate at which the rate of height change can occur. But such non-linear version of the energy model would not be viable in the route planning stage as the cost of travel between nodes is considered as individual components. However upon calculation of the path, a non-linear energy

model could provide a far more accurate method to defining the exact energy required by a UAV.

7 Report Methodology

7.1 Project Management and Organisation

To complete the objectives of this project while allowing sufficient time for each step careful project management was required. To produce a project plan the objectives of the project were considered and broken down into the elements that were required to complete each stage. The prerequisite requirements of each item in the project plan were then defined to ensure that each step could be completed when required. These requirements led to a framework for a project plan that only required estimation of the time required for each stage to be complete.

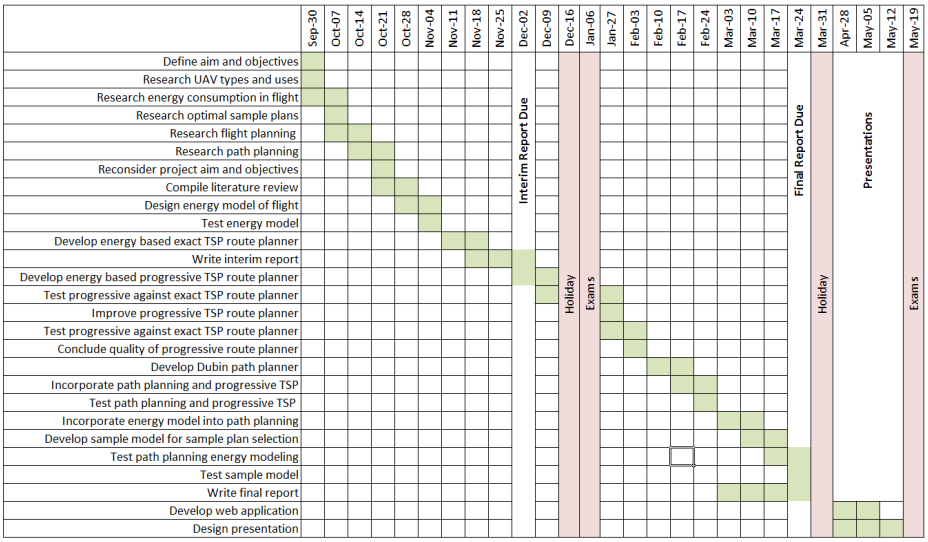


Figure 25: Gantt Chart of Project Plan

Figure 25 shows the project plan followed. The steps are on the left and the time which is allocated for completion indicated by coloured squares. This plan was not followed to the letter as some of the stages were more complicated than first imagined so a more gradual approach to developing the solution was required. These overruns were either due to more detailed literature analysis being required or continued trial and error on the part of programming the solution.

7.2 Writing the Report

The consideration and thought that went into composing ideas in the report often led to further analysis on the part of the code produced, it therefore seemed logical to

produce the report in a more adaptive manor. This resulted in designing a parametric report where calculations, results and figures were automatically updated upon each run.

The adaptive report is written in python as all other project elements were computed using python. The basis of the logic is a TexDocument class that has a number of methods that deal with adding basic elements to a LaTeX document. Within the main content document these methods can easily be called to define: sections, paragraphs, figures, equations and tables. All these elements are added to a working LaTeX document as they are interpreted. Upon completion of the report the compile method can be called which calls pdfTex and BibLaTeX to build the pdf and automatically open it to view. In addition to standard features this approach means that: the LaTeX document automatically remunerates figure labels if there is repetition to ensure there is no cross over and nomeclature and abreviations can be collected within a dictionary from any point within the report.

Producing the report in this way means that whatever change of code was made can easily result in an updated report. The drawback however is the compute time required to compile the report. This problem was levitated using cached results to computations. The function loadOrRun within the shared module of the code for this project deals with caching of results. If parameters are changes then the cache automatically recalculates the required data. Additionally all display logic is run on each compile so a change in plot style is quick and easy to implement without having to recomputed the cache.

7.3 Python Code

The code that has been used in this project is contained within this section. Comments have been added at frequent points in the code to show the logic behind what is being done. The modules detailed in this section are as follows:

- Shared
- Plane Energy
- Air Foil
- Latin Hypercube
- Travelling Plane
- Sample Model
- Dubins Path

Shared

Shared is a module that contains all the logic available that is required by all modules of the project and contains generic functions that could be used in all locations.

```
import os,re,csv,numpy,math,operator,itertools,math,operator,json,collections

#global debug variable
DEBUG = False

#global variables used in computations
MAX_LENGTH = 1000
LOWEST_NODE = 20
HIGHEST_NODE = 120
ALL_NODES = list(range(LOWEST_NODE,HIGHEST_NODE+1))
ALL_BETAS = [round(0.1*item,1) for item in range(1,5)]

def naturalKeys(text):
    '''
    alist.sort(key=natural_keys) sorts in human order
    http://nedbatchelder.com/blog/200712/human_sorting.html
    '''

    def atoi(text):
        return int(text) if text.isdigit() else text

    return [ atoi(c) for c in re.split('(\d+)', text) ]

def changeArray(array):
    """function that cahgnes the configuration of an array

    given input: [[a1,b1,c1],[a2,b2,c2]]
    returns: [[a1,a2],[b1,b2],[c1.c2]]
    """

    return [[float(array[j][i]) for j in range(len(array))] for i in range(le

def loadOrRun(filename,function,*args):
    """load run attempts to open the given filename and if this is possible r
    if the filename is not accessible for some reason loadRun runs the given
    the returned result is saved as a JSON
    """
    def loadJSON(filename):
        "saves the data object as a JSON string"
        with open(filename,"r") as openFile:
```

```
            data = json.loads(openFile.read())
        return data

    def saveJSON(filename,data):
        "saves the data object as a JSON string"
        with open(filename,"w") as openFile:
            openFile.write(json.dumps(data))
    try:
        data = loadJSON(filename)
    except FileNotFoundError:
        data = function(*args)
        saveJSON(filename,data)

    return data

def loadData(filename):
    "loads the data in the given file into a dictionary where the colo

    with open(filename,"r") as csvFile:
        csvReader = csv.reader(csvFile,dialect="excel")

        data = {}
        for i,row in enumerate(csvReader):
            if (i == 0):
                columbNames = []
                for item in row:
                    columbNames.append(item)
                    data[item] = []
            else:
                for j,item in enumerate(row):
                    if (len(item) > 0):
                        data[columbNames[j]].append(float(item))

    for key,value in data.items():
        if ((type(value) == list) and (len(value)==1)):
            data[key] = value[0]

    return data

def saveData(filename,data):
```

```
    """function to save the given data as a CSV file
    only writes the shortest data set in dictionary
    dictionary key is columb name and value is array
    """

    if (type(data) != dict):
        raise ValueError("The function provided did not return a single dictio
    elif not all([isinstance(data[key], collections.Iterable) for key in data
        try:
            for key,value in data.items():
                if (type(value) != list):
                    data[key] = [value]
        except Exception as exception:
            raise ValueError("The function returned a dictionary with values

    with open(filename, 'w', newline='') as csvfile:
        csvWriter = csv.writer(csvfile, dialect="excel")

        columbNames = []

        for key in data.keys():
            columbNames.append(key)

        csvWriter.writerow(columbNames)

        numberRows = max([len(data[key]) for key in columbNames])
        coloumbValues = [coloumbValue+[None]*(numberRows-len(coloumbValue)) f

        for i in range(numberRows):
            row = [item[i] for item in coloumbValues]
            csvWriter.writerow(row)

def calculateEnergy(distance,height,beta):
    """calculates and returns the energy required to travel between two point

    beta represents the distance climb ratio
    beta = 1: distance is equal weighting to height gain
    beta = 0: only height gain is considered
    """
```

```
    #check if height change is too great
    if (abs(height) > distance):
        return numpy.infty

    energy = beta*distance + height

    return max([0,energy])
```

Plane Energy

Plane energy is a module that is used to model the energy consumption of a plane. This module contains a number of global variables that define a default plane. A single class within this module is constructed using plane variables, this class is capable of computing the energy required for a number of flight manoeuvres and returning the energy coefficient and energy factor.

```
from shared import *
import numpy,plot,airFoil

#set standard environmental variables
gravity = 9.81
density = 1.19
viscosity = 1.82*10**-5

#set default plane variables
foilName = "naca23015-il"
mass = 0.64
wingSpan = 1.5
wingArea = 0.204
oswaldFactor = 0.7
minTurnRadius = 20

class PlaneEnergy():
  """PlaneEnergy is a class designed to similate flight manouvers"""

  def __init__(self, mass, wingSpan, wingArea, oswaldFactor, minTurnR
    "method to construct the class with plane variables"

    self.mass = mass
```

```
    self.minTurnRadius = minTurnRadius
    self.wingSpan = wingSpan
    self.wingArea = wingArea
    self.wingChord = wingArea/wingSpan
    self.oswaldFactor = oswaldFactor
    self.aspectRatio = wingSpan**2/wingArea
    self.flightVelocity = None

  def setAirFoil(self,foilName,reynoldsNumber=1000000):
    "method to set the airfoil type for the plane"

    self.foilName = foilName
    self.reynoldsNumber = reynoldsNumber
    self.airFoil = airFoil.AirFoil(foilName,reynoldsNumber)
    return self.setFlightData()

  def setFlightData(self):
    "method to set the attack angle for optimal range"

    #configure attack angles and coeficients from xfoil
    self.attackAngles = self.airFoil.getAoAs()
    self.cLifts = self.airFoil.getCLs()
    self.cZeroDrags = self.airFoil.getCDs()

    #set coeficient of drag including induced drag
    self.cDrags = self.cZeroDrags + (self.cLifts**2)/(numpy.pi*self.aspectRat
    #define the different sides of range equation for minimisation
    rangeEquation = abs(self.cLifts - numpy.sqrt(self.cZeroDrags*numpy.pi*sel
    #minimise range equation to select flight data for best range
    index = numpy.where(rangeEquation==min(rangeEquation))

    #set angle of attack, coeficient of lift and coeficient of drag
    self.attackAngle = self.attackAngles[index]
    self.cLift = self.cLifts[index]
    self.cDrag = self.cDrags[index]

    #if no flight velocity set configure flight velocity
    if not self.flightVelocity:
      self.setFlightVelocity()
      return self.flightVelocity
```

```
    #return the reynolds number
    return self.reynoldsNumber

  def checkReynoldsNumber(self):
    "method to check the configured reynolds number complies with the

    #if no veloicty use max range velocity
    if not self.flightVelocity:
      self.setFlightVelocity()

    #calculate the reynolds number from flight velocity
    calculatedReynoldsNumber = float((density*self.wingChord*self.flig
    newReynoldsNumber = self.airFoil.setReynoldsNumber(calculatedReyno

    #if reynolds number changed return new velocity
    if (newReynoldsNumber != self.reynoldsNumber):
      self.reynoldsNumber = newReynoldsNumber
      return self.setFlightData()
    else:
      return self.reynoldsNumber

  def setFlightVelocity(self,manualVelocity=None):
    "method to set the cruise velocity either manually or from angle

    #use given velocity else select optimal velocity
    if manualVelocity:
      self.flightVelocity = manualVelocity
    else:
      self.flightVelocity = numpy.sqrt((2*self.mass*gravity)/(density*

    #given the flight velocity is set test that reynolds number is ad
    return self.checkReynoldsNumber()

  def levelFlight(self,distance):
    "method to specifiy level flight and return energy required for m

    return 0.5*distance*density*self.cDrag*self.wingArea*self.flightV

  def climbingFlight(self,distance,height):
```

```
    "method to specifiy climbing flight and return the energy required for ma

    levelEnergy = self.levelFlight(distance)
    climbEnergy = self.mass*gravity*height
    return levelEnergy+climbEnergy

  def turningFlight(self,distance,turnRadius):
    "method to specifiy turning flight at given turning radius and return the

    if turnRadius < self.minTurnRadius:
      print("Minimum turning radius reached {}m".format(self.minTurnRadius))
      turnRadius = self.minTurnRadius

    loadFactor = numpy.sqrt(1+(self.flightVelocity**4)/(gravity*turnRadius)**
    return distance*(gravity*self.cDrag*loadFactor*self.mass)/self.cLift

  def climbingTurningFlight(self,distance,height,turnRadius):
    "method to specifiy turning and climbing flight at maximum rate of turn a

    levelEnergy = self.turningFlight(distance,turnRadius)
    climbEnergy = self.mass*gravity*height
    return levelEnergy+climbEnergy

  def getEnergyCoefficient(self):
    "method to return the energy coeficient beta"

    return float(self.levelFlight(1)/self.climbingFlight(0,1))

  def getEnergyFactor(self):
    "method to return the energy factor gamma"

    return float(self.climbingFlight(0,1))
```

Air Foil

Airfoil is a module that is used to obtain data on a number of airfoils using *airfoil-tools.com*. The class contained within this module is constructed with a foil name and Reynolds number and can return the variation of the lift and drag coefficients with changing angle of attack.

```
from shared import *
import numpy,re,urllib.request

class AirFoil():
  """class to load airfoil data and return the lift and drag values d

  def __init__(self, foilName, reynoldsNumber=1000000):
    "initiates the class by calling setFoilData that populates alpha,

    self.setFoilData(foilName,reynoldsNumber)

  def getAoAs(self):
    "returns all possible angles of attacks possible"

    return numpy.array(self.AoAs)

  def getCLs(self):
    "returns all values for the coeficient of lift"

    return numpy.array(self.CLs)

  def getCDs(self):
    "returns all values for the zero lift coeficient of drag"

    return numpy.array(self.CDs)

  def setReynoldsNumber(self,reynoldsNumber=1000000):
    "method to set reynolds number after constructing class"

    self.setFoilData(self.foilName,reynoldsNumber)
    return self.reynoldsNumber

  def setFoilData(self,foilName,reynoldsNumber):
    "returns a dictionary of the airfoils lift and drag coeficients a

    self.foilName = foilName
    self.reynoldsNumber = reynoldsNumber
    self.AoAs = []
    self.CDs = []
    self.CLs = []
```

```
#load list of foil names
with open("airfoils.txt") as openFile:
  airfoils = openFile.read().splitlines()

#check requested foil is in list
if (foilName not in airfoils):
  raise ValueError("Foil name {} does not exist".format(foilName))

#if reynolds number not in list use closest estimate
reynoldsNumbers = [50000,100000,200000,500000,1000000]
if (reynoldsNumber not in reynoldsNumbers):
  difReynoldsNumbers = [abs(reynoldsNumber-item) for item in reynoldsNumb
  self.reynoldsNumber = reynoldsNumbers[difReynoldsNumbers.index(min(difR

#access the data file
url = 'http://airfoiltools.com/polar/text?polar=xf-{}-{}'.format(self.foil
with urllib.request.urlopen(url) as webPage:
  content = webPage.read().splitlines()

#set not to record data until certain point
record = False

#cycle through lines of content to obtain alpha, CD and CL
for row in content:

  #split row by space characters
  row = str(row)
  rowList = re.split(r'\s*',row)

  #if recording is true and not line break line save values
  if (record and ("--" not in row)):
    self.AoAs.append(float(rowList[1]))
    self.CLs.append(float(rowList[2]))
    self.CDs.append(float(rowList[3]))

  #if columb headers found set recording to True
  if (("alpha" == rowList[1]) and ("CL" == rowList[2]) and ("CD" == rowLi
    record = True
```

```
        self.AoAs = numpy.array(self.AoAs)
        self.CDs = numpy.array(self.CDs)
        self.CLs = numpy.array(self.CLs)
```

Latin Hypercube

Latin hypercube is a module that calls MATLAB to connect with the code produced by Forrester et al., 2008 and return Latin hypercube sampling plans of any given number of nodes and number of dimensions. The results to these MATLAB calls are cached to reduce the time of subsequent calls.

```
from shared import *
import numpy,os,sys,csv,subprocess,time

def unitCube(numberOfNode):
    "function to return the nodes contained within a unit latin hypercub

    return samplePlan(3,numberOfNode)

def allUnitCubes():
    "function to calculate all unit cube latin hypercubes"

    for i in ALL_RESULTS:
        unitCube(i)

def samplePlan(numberDimension,numberOfNode):
    "function to load or compute latin hypercubes of given number of no

    numberDimension = int(numberDimension)
    numberOfNode = int(numberOfNode)
    matlabUpdate = "update_data.m"

    #if less than two nodes return None
    if (numberOfNode < 2):
        return None

    def updateMatlab(numberDimension,numberOfNode):
        """function to run the required matlab script that updates all re
```

```
    assumes that the function to update matlab is stored in a folder called m
    """

    #define matlab update file
    fileContents = ""
    fileContents+="output = bestlh({},{},1,1);\n".format(numberOfNode,numberD
    fileContents+="csvwrite('latin_hypercube_{}_{}.csv',output);".format(numb

    #write update file for matlab to call
    updatePath = "matlab/update_data.m"
    with open(updatePath,"w") as openFile:
      openFile.write(fileContents)

    #define command to call matlab in current working directory and run updat
    currentDirectory = os.getcwd()
    matlabCommand = 'cd {}, run {}, exit'.format(currentDirectory,updatePath)

    #run command with no windows or spash screen
    subprocess.check_call(['matlab', '-wait', '-automation', '-nosplash', '-n

  def loadMatlab(filePath):
    "function to load the data created by MATLAB"

    #for all rows in the output file
    nodes = []
    with open(filePath,"r") as csvFile:
      csvReader = csv.reader(csvFile,dialect="excel")
      for row in csvReader:
        nodes.append([float(item) for item in row if len(item)>0])

    #return numpy array of node coordinates
    return [numpy.array(node) for node in nodes]

  filePath = "matlab/latin_hypercube_{}_{}.csv".format(numberDimension,number
  try:
    #attempt to open the required hypercube
    open(filePath)
  except FileNotFoundError:
    #if file not found update matlab and wait for a second
    updateMatlab(numberDimension,numberOfNode)
```

```
    time.sleep(1)

  return loadMatlab(filePath)

def sampleSpace(lengths,numberOfNode):
  "function to return a latin hypercube scaled to the required sample

  #define number of dimensions from length of lengths
  dimensions = len(lengths)
  #obtain the required number of nodes of given dimensions
  nodes = samplePlan(dimensions,numberOfNode)
  #create numpy array of lengths for easy dot product multiplication
  lengths = numpy.array(lengths)
  #preallocate memory for scaled nodes
  scaledNodes = numpy.zeros([numberOfNode,dimensions])

  #for all nodes populate scaled nodes
  for i,node in enumerate(nodes):
    scaledNodes[i] = node*lengths

  return scaledNodes

if (__name__ == "__main__"):
  allUnitCubes()
```

Travelling Plane

Travelling plane is a module that is used to calculate the least cost route through given nodes. The module contains both the exact all routes approach and the progressive travelling plane approach.

```
from shared import *
import numpy,itertools

def progressiveRoute(nodes,beta,nodesPerRoute=4):
  """function to progressively work through the given nodes and
  return an approximation of the best cost route"""

  #define total nodes in subset
```

```
totalRouteNodes = nodesPerRoute*2

#set all nodes as numpy arrays then set energy matrix
numberNode = len(nodes)
nodes = [numpy.array(node) for node in nodes]
setEnergyMatrix(nodes,beta)

#define list of nodes sorted by vertical location
sortedIndexs = sorted(list(range(len(nodes))), key = lambda x:nodes[x][2])

#assign start and end points for total route
routeA = [sortedIndexs[0]]
routeB = [sortedIndexs[1]]

#work through all nodes in steps of two
for i in range(0,numberNode,2):

  #select last indexs of both routes as base indexs
  indexA = routeA[-1]
  indexB = routeB[-1]

  #consider nodes above current
  currentIndexs = sortedIndexs[:i+totalRouteNodes]

  #remove nodes already in route
  for index in routeA[:-1]:
    currentIndexs.remove(index)
  for index in routeB[:-1]:
    currentIndexs.remove(index)

  #if there are 3 or less nodes left calculate the exact route
  if (len(currentIndexs) <= 3):
    #find all possible routes between last node in route a and first in rou
    routes = routePermutations(currentIndexs,indexA,indexB)
    #compute costs for each route
    costs = [routeCost(route) for route in routes]
    #select cheapest cost
    bestCost = min(costs)
    #select corresponding route
    bestRoute = routes[costs.index(bestCost)]
```

```
    #for all items in route aside from start and end nodes add them
    for index in bestRoute[1:-1]:
      routeA.append(index)
    #break the for loop as route complete
    break

#compute the index of the highest node
highestIndex = sorted(currentIndexs, key=lambda x:nodes[x][2])[-1]

#calculate all possible route combinations using even number of cu
possibleRoutes = doubleRoutePermutations(currentIndexs[:2*(len(cu

#assign a default best route and cost to
bestRoute = possibleRoutes[0]
bestCost = numpy.infty

#for all possible routes check if least cost
for route in possibleRoutes:

  #define up route A (plus route to highest node in subset)
  currentRouteA = list(route[0])+[highestIndex]
  #define down route B
  currentRouteB = [i for i in reversed(route[1])]

  #compute cost of both routes
  costA = routeCost(currentRouteA)
  costB = routeCost(currentRouteB)
  cost = sum([costA,costB])

  #if cost is improvement assign best route and best cost
  if (cost<bestCost):
    bestRoute = route
    bestCost = cost

#decompose best route into A and B
bestRouteA,bestRouteB = bestRoute

#if the best routes are more than inity length add second nodes to
if (max([len(bestRouteA),len(bestRouteB)])>1):
  routeA.append(bestRouteA[1])
```

```
        routeB.append(bestRouteB[1])

    #combine routes A and B to form a single best route
    routeB.reverse()
    bestRoute = routeA+routeB
    bestCost = routeCost(bestRoute,True)

    return bestRoute,bestCost

def allRoutes(beta,nodes,startIndex=None,endIndex=None):
    "function to compute and return all possible route combinations and the res

    #define indexs of all nodes given
    numberNodes = len(nodes)
    nodeIndexs = list(range(0,numberNodes))

    #if start node defined remove from index list
    if (startIndex != None):
        nodeIndexs.pop(startIndex)
        startIndex = [startIndex]
    else:
        startIndex = []

    #if end node defined remove from index list
    if (endIndex != None):
        nodeIndexs.pop(endIndex)
        endIndex = [endIndex]
    else:
        endIndex = []

    #set energy matrix from all nodes
    nodes = [numpy.array(node) for node in nodes]
    setEnergyMatrix(nodes,beta)

    #for all permutations append route and cost
    routes,costs = [],[]
    for route in itertools.permutations(nodeIndexs):
        route = startIndex+list(route)+endIndex
        cost = routeCost(route,True)
        routes.append(route)
```

```
        costs.append(cost)

    return routes,costs

def orderNodes(nodes,order,loop=False):
    "function to order the given nodes and return along with the final r

    #order nodes by order given
    orderedNodes = [nodes[index] for index in order]

    #if loop required append first node at end
    if loop: orderedNodes.append(nodes[order[0]])

    return orderedNodes

def routeCost(route,loop=False):
    "calculates the cost of a route given a route (sequence of nodes)"

    cost = 0
    #for all connections between nodes append the cost of the path
    for i in range(len(route)-1):
        nodeIndexFrom = route[i]
        nodeIndexTo = route[i+1]
        cost += energyMatrix[nodeIndexFrom][nodeIndexTo]

    #if loop required add cost of path back to start node
    if loop:
        nodeIndexFrom = route[-1]
        nodeIndexTo = route[0]
        cost += energyMatrix[nodeIndexFrom][nodeIndexTo]

    return cost

def setEnergyMatrix(nodes,beta):
    "returns a matrix of energy costs to navigate the given node locatio

    global energyMatrix
    numberNodes = len(nodes)
    energyMatrix = numpy.zeros([numberNodes,numberNodes])
```

```
  for i in range(numberNodes):
    for j in range(numberNodes):
      vector = nodes[j]-nodes[i]
      distance = numpy.linalg.norm(vector)
      height = vector[2]
      energyMatrix[i][j] = calculateEnergy(distance,height,beta)


def routePermutations(indexs,startIndex=None,endIndex=None):
  """function to return all possible route permutations given a start and end

  #if start node defined remove from indexs
  if (startIndex != None):
    indexs.remove(startIndex)
    startIndex = [startIndex]
  else:
    startIndex = []

  #if end node defined remove from indexs
  if (endIndex != None):
    indexs.remove(endIndex)
    endIndex = [endIndex]
  else:
    endIndex = []

  #compute all possible permutations
  permutations = []
  for route in itertools.permutations(indexs):
    permutations.append(startIndex+list(route)+endIndex)

  return permutations

def doubleRoutePermutations(indexs,nodeA=None,nodeB=None):
  """function to display the possible permutation sets for 2 routes
  starting at node index nodeA and node index nodeB in set indexs"""

  #define the number nodes and route length given the node indexs
  numberNodes = len(indexs)
  routeLength = numberNodes//2
```

```
    #if the number of nodes is not even rasise an error
    if (numberNodes%2 != 0):
      raise ValueError("N needs to be an even number of indexs, {} is no

    #for all potential permutations of length N/2 in N indexs
    permutationA,permutationB = [],[]
    for item in itertools.permutations(indexs,routeLength):
      #if nodeA is the start node or nodeA is none append to permuatonA
      if ((nodeA == item[0]) or (nodeA == None)):
        permutationA.append(item)
      #if nodeB is the start node or nodeB is none append to permuatonB
      elif ((nodeB == item[0]) or (nodeB == None)):
        permutationB.append(item)

    def checkRoutes(routeA,routeB):
      """checks the given routes to see if any indexs are repeated

      returns True if exclusive
      returns False if any repetition
      """
      for node in routeA:
        if (node in routeB):
          return False
      return True

    #for all combinations of sub routes
    permutations = []
    for routeA in permutationA:
      for routeB in permutationB:
        #if the combination is exclusive append to final permutations
        if checkRoutes(routeA,routeB):
          permutations.append([routeA,routeB])

    return permutations

  def routeData(nodes,beta,loop=False):
    "function to return the route data of the given node ordering"

    distances,heights,costs = [],[],[]
```

```
def distanceHeightCost(nodeFrom,nodeTo):
  "returns the distance,height and cost for travel between nodes"

  vector = nodes[i+1]-nodes[i]
  distance = numpy.linalg.norm(vector)
  height = vector[2]
  cost = calculateEnergy(distance,height,beta)

  return distance,height,cost

for i in range(len(nodes)-1):
  nodeFrom = nodes[i]
  nodeTo = nodes[i+1]
  distance,height,cost = distanceHeightCost(nodeFrom,nodeTo)

  distances.append(distance)
  heights.append(height)
  costs.append(cost)

if loop:
  nodeFrom = nodes[-1]
  nodeTo = nodes[0]
  distance,height,cost = distanceHeightCost(nodeFrom,nodeTo)

  distances.append(distance)
  heights.append(height)
  costs.append(cost)

heights = [abs(height) for height in heights]
glideHeight = sum([height for i,height in enumerate(heights) if (costs[i] =
glideDistance = sum([distance for i,distance in enumerate(distances) if (co

totalHeight = sum(heights)
totalDistance = sum(distances)
totalCost = sum(costs)

data = {
  "cost":totalCost,
  "height":totalHeight,
  "distance":totalDistance,
```

```
    "glideHeight":glideHeight,
    "glideDistance":glideDistance
  }

  return data
```

Sample Model

Sample model is a module that is used to compute models of the energy cost of a routes given different scenarios. This module enables the route planning for atmospheric data collection to be done from the requirement of total energy consumed.

```
from shared import *
import numpy,time,scipy.optimize,latinHypercube,travellingPlane,dubinP

def returnNodes(beta,xLength,yLength,zLength,totalEnergy):
  """using the calculated models this function takes the:

  Research volume as defined by xLength,yLength and zLength
  The totalEnergy that the UAV batteries contain

  And returns the number of nodes in the optimal latin hypercube"""

  #load the results to the model calculations
  filename = "results/model_variables.csv"
  variables = loadData(filename)

  #collect nodes and betas
  numberNodes = variables["N"]
  betas = variables["A"]

  #for all numbers of nodes
  for numberNode in ALL_NODES:

    #find all indexs of nodes
    nodeIndexs = [i for i in range(len(numberNodes)) if numberNode ==
    #find the closes beta value
    betaDiffs = [abs(beta-betas[i]) for i in nodeIndexs]
    #return the index of closest value
```

```
    index = nodeIndexs[betaDiffs.index(min(betaDiffs))]

    #set number of nodes using index to ensure correct
    N = variables["N"][index]
    A = variables["A"][index]

    #set the energy model parameters using index
    energyParameters = [
      variables["ma"][index],
      variables["ea"][index],
      variables["md"][index],
      variables["ed"][index],
      variables["mz"][index],
      variables["ez"][index],
      variables["c"][index]
      ]

    #calculate distance, height and thus energy
    energy = energyModel(energyParameters,xLength,yLength,zLength)

    #if energy is more than total energy
    if (energy > totalEnergy):

      #if first row flight not possible therefore return None
      if (numberNode == ALL_NODES[0]):
        return None

      #take the previous value for number of nodes
      N = variables["N"][index-1]

      #end loop
      break

  if (N == ALL_NODES[-1]):
    return None

  # print(A,N,energy)
  return int(N)

def computeResults(beta,numberNode):
```

```
    "function to fully investigate the resulting routes of a latin hyper

    testCases = latinHypercube.sampleSpace([MAX_LENGTH,MAX_LENGTH,MAX_LE

    filename = "results/route_path_lengths_{:0.1f}_{:0.0f}.csv".format(

    try:
      data = loadData(filename)
    except FileNotFoundError:
      data = {"X":[],"Y":[],"Z":[],"Energy":[]}
      for testCase in testCases:

        [X,Y,Z] = testCase
        nodes = latinHypercube.sampleSpace([X,Y,Z],numberNode)
        route,energy = travellingPlane.progressiveRoute(nodes,beta)
        orderedNodes = travellingPlane.orderNodes(nodes,route)
        routeData = travellingPlane.routeData(orderedNodes,beta)

        data["X"].append(X)
        data["Y"].append(Y)
        data["Z"].append(Z)
        data["Energy"].append(energy)

      saveData(filename,data)

    return data

def energyModel(parameters,x,y,z):
    ma,ea,md,ed,mz,ez,c = parameters

    return ma*(x*y)**ea + md*numpy.abs(x-y) + mz*z**ez + c

def returnModel(beta,numberNode):
    "computes a model for length width and height given the number of n

    def collectResults():
      "function to collect all results of the space analysis for model

      data = computeResults(beta,numberNode)
```

```
    X = numpy.array(data["X"])
    Y = numpy.array(data["Y"])
    Z = numpy.array(data["Z"])
    E = numpy.array(data["Energy"])

    energyParameters = buildModel(E,X,Y,Z)
    [ma,ea,md,ed,mz,ez,c] = energyParameters

    return energyParameters

  def buildModel(output,x,y,z):
    """function to build a model from a number of parameters"""

    def residuals(parameters,output,x,y,z):

      return output-energyModel(parameters,x,y,z)

    #preliminary model parameters
    [ma,ea,md,ed,mz,ez,c] = [0.05,0.8,1,1,0.8,1.1,100]
    parameters = numpy.array([ma,ea,md,ed,mz,ez,c])
    #model arguments
    arguments = (output,x,y,z)
    #compute least squares regression
    result,sucess = scipy.optimize.leastsq(residuals,parameters,arguments)

    return result

  return collectResults()

def getEquation(beta=None,numberNode=None):
  "function to return the equation of either height or distance for the given

  if (numberNode and beta):

    filename = "results/model_variables.csv"
    variables = loadData(filename)

    numberNodes = variables["N"]
    betas = variables["A"]
```

```
    nodeIndexs = [i for i in range(len(numberNodes)) if numberNode ==
    betaDiffs = [abs(beta-betas[index]) for index in nodeIndexs]
    index = nodeIndexs[betaDiffs.index(min(betaDiffs))]

    #set the energy model parameters
    [ma,ea,md,ed,mz,ez,c] = [
      variables["ma"][index],
      variables["ea"][index],
      variables["md"][index],
      variables["ed"][index],
      variables["mz"][index],
      variables["ez"][index],
      variables["c"][index]
      ]
    equation = "E={:0.2f}(xy)^{{{:0.3f}}}+{:0.2f}abs(x-y)+{:0.2f}z^{{
  else:
    #if there are no beta or numberNode set then return the stock equa
    [ma,ea,md,ed,mz,ez,c] = ["ma","ea","md","ed","mz","ez","c"]
    equation = r"E={}\cdot (xy)^{{{}}}+{}\cdot abs(x-y)+{}\cdot z^{{{

  return equation

def collectAllResults():
  "collects all results for the length width height model"

  #define nodes and betas from shared variables
  numberNodes = ALL_NODES
  betas = ALL_BETAS

  variables = {
    "A"    :[],
    "N"    :[],
    "ma"   :[],
    "ea"   :[],
    "md"   :[],
    "ed"   :[],
    "mz"   :[],
    "ez"   :[],
    "c"    :[]
  }
```

```
  for numberNode in numberNodes:
    for beta in betas:
      variables["A"].append(round(beta,1))
      variables["N"].append(int(numberNode))
      energyParameters = returnModel(beta,numberNode)

      [ma,ea,md,ed,mz,ez,c] = energyParameters
      variables["ma"].append(ma)
      variables["ea"].append(ea)
      variables["md"].append(md)
      variables["ed"].append(ed)
      variables["mz"].append(mz)
      variables["ez"].append(ez)
      variables["c"].append(c)

  filename = "results/model_variables.csv"
  saveData(filename,variables)

def computeAllResults(newSample):
  """function to compute all length width height results possible

  this function uses a csv sample planner so can be run on multiple interprit
  thus allowing full processor utilisation
  """

  filename = "sample_plan.csv"
  if newSample:
    data = {
      "numberNode":[ALL_NODES[i//len(ALL_BETAS)] for i in range(len(ALL_NODES
      "beta":ALL_BETAS*len(ALL_NODES)*len(ALL_BETAS)
      }
    saveData(filename,data)

  while not newSample:
    #load the test case data
    data = loadData(filename)

    #break the while loop if there are no tests left to compute
    if ((len(data["beta"]) == 0) or (len(data["numberNode"]) == 0)):
```

```
            break

        #remove test values from file data
        beta = round(float(data["beta"].pop(0)),1)
        numberNode = int(data["numberNode"].pop(0))

        #print the current test case
        print(beta,numberNode)

        try:
          #attempt to save the sample file with the current test cases ren
          saveData(filename,data)
          try:
            #attempt to compute the model with given parameters
            returnModel(beta,numberNode)
          except Exception as exception:
            #if there is a problem print the error
            print("Problem with node: {} a: {} \n {}".format(numberNode,be
            #return the current test case to the sample plan
            data["beta"].insert(0,beta)
            data["numberNode"].insert(0,numberNode)
            saveData(filename,data)
        except PermissionError:
          print("Sleep")
          time.sleep(20)


if (__name__ == "__main__"):
    computeAllResults(True)
```

Dubins Path

Dubins path is a module used to compute the shortest distance Dubins paths either between two points with start and end directions defined or using the Dubins Path class the total path through a number of nodes. This module is not fully commented as the geometric logic is from Giese, 2012 and there is a lot of lines of code to explain.

```
from shared import *
```

```
import numpy,plot

def linePath(startPoint,endPoint,distanceOnly=True):
  "draws a line with the given start and end points"

  startPoint = numpy.array(startPoint)
  endPoint = numpy.array(endPoint)
  distance = numpy.linalg.norm(endPoint - startPoint)

  if distanceOnly:
    return distance

  x = [startPoint[0],endPoint[0]]
  y = [startPoint[1],endPoint[1]]
  z = [startPoint[2],endPoint[2]]

  mode = "straight"
  height = endPoint[2]-startPoint[2]
  path = [x,y,z]

  return (mode,distance,height,path)

def arcPath(centre,startPoint,endPoint,direction,distanceOnly=True):
  "draws an arc with the given centre and radius given the start and end poin

  centre = numpy.array(centre)
  startPoint = numpy.array(startPoint)
  endPoint = numpy.array(endPoint)

  #2D problem
  radiusA = numpy.linalg.norm(startPoint[:2]-centre)
  radiusB = numpy.linalg.norm(endPoint[:2]-centre)
  radius = max(radiusA,radiusB)

  [X1,Y1] = startPoint[:2] - centre
  [X2,Y2] = endPoint[:2] - centre
  startAngle = numpy.arctan2(Y1,X1)
  endAngle = numpy.arctan2(Y2,X2)

  if ((startAngle<endAngle) and direction == "R"):
```

```
    startAngle += 2*numpy.pi
  elif ((startAngle>endAngle) and direction == "L"):
    endAngle += 2*numpy.pi

  theta = abs(startAngle - endAngle)
  distance = radius*theta

  if distanceOnly:
    return distance

  #compute path
  N = 50
  thetas = numpy.linspace(startAngle,endAngle,N)
  x = centre[0] + radius*numpy.cos(thetas)
  y = centre[1] + radius*numpy.sin(thetas)

  #3D results
  z = numpy.linspace(startPoint[2],endPoint[2],N)
  mode = "turn"
  height = endPoint[2]-startPoint[2]
  distance = numpy.sqrt(distance**2+height**2)
  path = [x,y,z]

  return (mode,distance,height,path)

def tangentLines(centre1,centre2,radius,pathType):
  """returns the tangent points between two circles with given orienta
  logic from document: http://gieseanw.files.wordpress.com/2012/10/du
  """

  #assign both radius to the same
  radius1 = radius2 = radius

  #convert to numpy array for calculation
  centre1 = numpy.array(centre1)
  centre2 = numpy.array(centre2)

  #compute connecting vector
  vector = centre1 - centre2
  distance = numpy.linalg.norm(vector)
```

```
#define tangent points based on configuration
if (pathType == "LSL"):
  ratio = (radius2-radius1)/distance
  if ((ratio < -1) or (ratio > 1)):
    if DEBUG: print("Path Type {} is not possible for inputs".format(pathTy
    return None,None
  angle = numpy.arccos(ratio)
  normal = computeVector(vector,angle)
  point1 = centre1 + normal*radius1
  point2 = centre2 + normal*radius2
elif (pathType == "RSR"):
  ratio = -(radius2-radius1)/distance
  if ((ratio < -1) or (ratio > 1)):
    if DEBUG: print("Path Type {} is not possible for inputs".format(pathTy
    return None,None
  angle = numpy.arccos(ratio)
  normal = computeVector(-vector,angle)
  point1 = centre1 + normal*radius1
  point2 = centre2 + normal*radius2
elif (pathType == "LSR"):
  ratio = -(radius2+radius1)/distance
  if ((ratio < -1) or (ratio > 1)):
    if DEBUG: print("Path Type {} is not possible for inputs".format(pathTy
    return None,None
  angle = numpy.arccos(ratio)
  normal = computeVector(vector,angle)
  point1 = centre1 + normal*radius1
  point2 = centre2 - normal*radius2
elif (pathType == "RSL"):
  ratio = (radius2+radius1)/distance
  if ((ratio < -1) or (ratio > 1)):
    if DEBUG: print("Path Type {} is not possible for inputs".format(pathTy
    return None,None
  angle = numpy.arccos(ratio)
  normal = computeVector(-vector,angle)
  point1 = centre1 + normal*radius1
  point2 = centre2 - normal*radius2

if (all(numpy.isnan(point1)) or all(numpy.isnan(point2))):
```

```
        if DEBUG: print("Path Type {} is not possible for inputs".format(p
        return None,None

    return point1,point2

def tangentCircles(centre1,centre2,radius,pathType):
    """returns the tangent points between three circles
    logic from document: http://gieseanw.files.wordpress.com/2012/10/du
    """

    #convert to numpy array for calculation
    centre1 = numpy.array(centre1)
    centre2 = numpy.array(centre2)

    #compute vector between circle centres and calculate length
    vector1 = centre2-centre1
    length = numpy.linalg.norm(vector1)

    #if length greater than 4 radius return None
    if (length >= 4*radius):
        if DEBUG:
            print("Circle centres are too far apart")
        return None,None,None

    #calculate angle to third circle depending on orientation
    if pathType == "LRL":
        theta = numpy.arccos(length/(4*radius))
    elif pathType == "RLR":
        theta = -numpy.arccos(length/(4*radius))

    #compute vector from first circle to third circle
    vector2 = computeVector(vector1,theta)
    centre3 = centre1 + vector2*2*radius

    #compute vector from second circle to third circle
    vector3 = centre3 - centre2
    vector3 = computeUnitVector(vector3)

    #calculate locations of tangents
    startTangent = centre1 + vector2*radius
```

```
    endTangent = centre2 + vector3*radius

    return startTangent,endTangent,centre3

def computeVector(vector,angle):
    "computes and returns the normal vector given a vector and angle"

    cosAngle = numpy.cos(angle)
    [v_x,v_y] = vector
    n_x = (v_x * cosAngle) - (v_y * numpy.sqrt(1-cosAngle**2))
    n_y = (v_x * numpy.sqrt(1-cosAngle**2)) + (v_y * cosAngle)
    normal = numpy.array([n_x,n_y])
    normal = computeUnitVector(normal)
    return normal

def computeUnitVector(vector):
    "computes and returns the unit vector of the given vector: vector"

    normal = numpy.linalg.norm(vector)
    if (normal == 0):
        return numpy.array([0]*len(vector))
    else:
        return vector/normal

def computeCentre(point,direction,radius,orientation):
    """computes the centre point of a circle given:
    point - the coordinates of a point that lies on the circumfrence of a circl
    direction - the direction of heading that is tangential to the circle at th
    radius - the radius of the circle
    orientation - weather the circle is left or right turn
    """

    [x,y] = point
    [d_x,d_y] = direction

    point = numpy.array(point)
    direction = numpy.array(direction)

    if (orientation == "L"):
        normal = computeVector(direction,numpy.pi/2)
```

```
    centre = point + radius*normal
  elif (orientation == "R"):
    normal = computeVector(-direction,numpy.pi/2)
    centre = point + radius*normal

  return centre

def dubinDistance(startPoint,startDirection,endPoint,endDirection,radi
  """ computes the lengths of any of the 6 options of dubin paths:
  ['RSR','LSL','RSL','LSR','RLR','LRL'] and returns details of each st
  """

  distances = []
  if (pathType in ['RSR','LSL','RSL','LSR']):

    startCentre = computeCentre(startPoint[:2],startDirection[:2],radi
    endCentre = computeCentre(endPoint[:2],endDirection[:2],radius,pat
    startTangent,endTangent = tangentLines(startCentre[:2],endCentre[:

    if (None in [startTangent,endTangent]):
      distances = None
    else:
      distances.append(arcPath(startCentre,startPoint,startTangent,pat
      distances.append(linePath(startTangent,endTangent))
      distances.append(arcPath(endCentre,endTangent,endPoint,pathType

  elif (pathType in ['RLR','LRL']):

    startCentre = computeCentre(startPoint[:2],startDirection[:2],radi
    endCentre = computeCentre(endPoint[:2],endDirection[:2],radius,pat
    startTangent,endTangent,middleCentre = tangentCircles(startCentre

    if None in [startTangent,endTangent,middleCentre]:
      distances = None
    elif all([round(startCentre[i],2) == round(endCentre[i],2) for i i
      distances.append(arcPath(startCentre,startPoint,endPoint,pathTyp
    else:
      distances.append(arcPath(startCentre,startPoint,startTangent,pat
      distances.append(arcPath(middleCentre,startTangent,endTangent,pa
      distances.append(arcPath(endCentre,endTangent,endPoint,pathType
```

```
    else:
        raise ValueError("Path type {} is not in list ['RSR','LSL','RSL','LSR','R

    return distances

def dubinPath(startPoint,startDirection,endPoint,endDirection,radius,pathType
    """computes the results of any of the 6 options of dubin paths:
    ['RSR','LSL','RSL','LSR','RLR','LRL']
    """

    if not distances:
        distances = dubinDistance(startPoint,startDirection,endPoint,endDirection

    startPoint = list(startPoint) + [0]*(3-len(startPoint))
    startDirection = list(startDirection) + [0]*(3-len(startDirection))
    endPoint = list(endPoint) + [0]*(3-len(endPoint))
    endDirection = list(endDirection) + [0]*(3-len(endDirection))

    height = endPoint[2]-startPoint[2]

    results = []
    if (pathType in ['RSR','LSL','RSL','LSR']):

        startCentre = computeCentre(startPoint[:2],startDirection[:2],radius,path
        endCentre = computeCentre(endPoint[:2],endDirection[:2],radius,pathType[2
        startTangent,endTangent = tangentLines(startCentre[:2],endCentre[:2],radi

        if (None in [startTangent,endTangent]):
            results = None
        else:
            startTangent = [i for i in startTangent]+[startPoint[2]+height*distance
            endTangent = [i for i in endTangent]+[endPoint[2]-height*distances[-1]/
            results.append(arcPath(startCentre,startPoint,startTangent,pathType[0],
            results.append(linePath(startTangent,endTangent,False))
            results.append(arcPath(endCentre,endTangent,endPoint,pathType[2],False)

    elif (pathType in ['RLR','LRL']):

        startCentre = computeCentre(startPoint[:2],startDirection[:2],radius,path
        endCentre = computeCentre(endPoint[:2],endDirection[:2],radius,pathType[2
```

```
        startTangent,endTangent,middleCentre = tangentCircles(startCentre

        if None in [startTangent,endTangent,middleCentre]:
            results = None
        elif all([round(startCentre[i],2) == round(endCentre[i],2) for i i
            results.append(arcPath(startCentre,startPoint,endPoint,pathType
        else:
            startTangent = [i for i in startTangent]+[startPoint[2]+height*
            endTangent = [i for i in endTangent]+[endPoint[2]-height*distanc
            results.append(arcPath(startCentre,startPoint,startTangent,pathT
            results.append(arcPath(middleCentre,startTangent,endTangent,path
            results.append(arcPath(endCentre,endTangent,endPoint,pathType[2]
    else:
        raise ValueError("Path type {} is not in list ['RSR','LSL','RSL',

    return results

def bestPath(startPoint,startDirection,endPoint,endDirection,radius):
    "computes the length of the best dubin path and returns route of the

    bestResults = None
    bestDistances = [numpy.infty]

    #cycle through options calculating distance
    for pathType in ['RSR','LSL','RSL','LSR','RLR','LRL']:

        distances = dubinDistance(startPoint,startDirection,endPoint,endD

        if (distances and (sum(distances) < sum(bestDistances))):
            bestPath = pathType
            bestDistances = distances

    return dubinPath(startPoint,startDirection,endPoint,endDirection,rad

class DubinPath():
    """DubinPath is a class to enable calculation and plotting of the m
    a number of nodes
    For each additional node DubinPath adds the energy distance and path
    """
```

```
def __init__(self, radius, planeModel=None):

  self.planeModel = planeModel
  self.radius = radius
  self.distance = 0
  self.energy = 0
  self.nodes = []
  self.lastNodes = [None,None]

  self.x,self.y,self.z = [],[],[]

def addNode(self,node):
  "method to add a node to the route"

  self.nodes.append(numpy.array(node))
  nodeCount = len(self.nodes)
  radius = self.radius

  if (nodeCount>2):
    previousNode = self.nodes[-3]
    startNode = self.nodes[-2]
    endNode = self.nodes[-1]
    startDirection = startNode - previousNode
    endDirection = endNode - startNode

    results = bestPath(startNode,startDirection,endNode,endDirection,self.r

    for result in results:
      mode,distance,height,path = result

      self.distance += distance
      self.x += [i for i in path[0]]
      self.y += [i for i in path[1]]
      self.z += [i for i in path[2]]

      if self.planeModel:
        if (mode == "straight"):
          self.energy += self.planeModel.climbingFlight(distance,height)
        elif (mode == "turn"):
          self.energy += self.planeModel.climbingTurningFlight(distance,hei
```

```
            else:
              raise ValueError("The mode of flight needs to be straight

        elif (nodeCount>1):
          self.lastNodes[1] = node
        elif (nodeCount>0):
          self.lastNodes[0] = node

    def makeLoop(self):
      "method to complete the paths loop"

      for i in range(len(self.lastNodes)):
        if self.lastNodes[i] != None:
          self.addNode(self.lastNodes[i])
          self.lastNodes[i] = None

    def getDistance(self):
      "method to return the distance of the path"

      self.makeLoop()
      return self.distance

    def getPath(self):
      "method to return the points of the path "

      self.makeLoop()
      return [self.x,self.y,self.z]

    def getEnergy(self):
      "returns the energy required to navigate the given path"

      self.makeLoop()
      return self.energy
```

8 References

[A. Forrester, 2009] A. Forrester, A. K. (2009). Recent advances in surrogate-based optimization. *Progress in Aerospace Sciences*, 45:50–79.

[Al-Sabban et al., 0] Al-Sabban, W. H., Gonzalez, L. F., and Smith, R. N. (0). Wind-energy based path planning for unmanned aerial vehicles using markov decision processes. *Journal*.

[Asselin, 1997] Asselin, M. (1997). *Introduction to Aircraft Performance*. American Institute of Aeronautics and Astronautics.

[Bigg et al., 1976] Bigg, N. L., Lloyd, E. K., and Wilson, R. J. (1976). *Graph Theory: 1736-1936*. Oxford University Press.

[Boissonnat et al., 1993] Boissonnat, J.-D., Cérézo, A., and Leblond, J. (1993). *Shortest paths of bounded curvature in the plane*. Springer.

[Bonin, 2011] Bonin, T. A. (2011). *Development and Initial Applications of the SMARTSonde for Meteorological Research*. PhD thesis, University of Oklahoma.

[Chakrabarty and Langelaan, 2009] Chakrabarty, A. and Langelaan, J. W. (2009). Energy maps for long-range path planning for small-and micro-uavs. In *AIAA Guidance, Navigation and Control Conference, AIAA Paper*, volume 6113.

[Chitsaz and LaValle, 2007] Chitsaz, H. and LaValle, S. M. (2007). Time-optimal paths for a dubins airplane. In *Decision and Control, 2007 46th IEEE Conference on*, pages 2379–2384. IEEE.

[De Berg et al., 2010] De Berg, M., Van Nijnatten, F., Sitters, R., Woeginger, G. J., and Wolff, A. (2010). The traveling salesman problem under squared euclidean distances. *arXiv preprint arXiv:1001.0236*.

[Dubins, 1957] Dubins, L. E. (1957). On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of mathematics*, 79(3):497–516.

[Forrester et al., 2008] Forrester, A., Sóbester, A., and Keane, A. (2008). *Engineering design via surrogate modelling: a practical guide*. John Wiley & Sons.

[Geographic, 2013] Geographic, N. (2013). Encyclopedic entry - atmosphere. `http://education.nationalgeographic.co.uk/education/encyclopedia/atmosphere/`.

[Giese, 2012] Giese, A. (2012). A comprehensive, step-by-step tutorial on computing dubins curves.

[Held et al., 1984] Held, M., Hoffman, A. J., Johnson, E. L., and Wolfe, P. (1984). Aspects of the traveling salesman problem. *IBM journal of Research and Development*, 28(4):476–486.

[Intelligence et al., 1996] Intelligence, D. D., Security Doctrine, D., and Instructions (1996). Uav class categories. `http://www.fas.org/irp/doddir/dod/jp3-55_1/3-55_1c1.htm`.

[Langelaan, 2007] Langelaan, J. W. (2007). Long distance/duration trajectory optimization for small uavs. In *AIAA Guidance, Navigation and Controls Conference*.

[Le Ny, 2008] Le Ny, J. (2008). *Performance optimization for unmanned vehicle systems*. PhD thesis, Massachusetts Institute of Technology.

[M. Price and Curran, 2006] M. Price, S. R. and Curran, R. (2006). An integratedsystems engineering approach to aircraft design. *Progress inAerospace Sciences*, 42:331–376.

[McGee et al., 2005] McGee, T. G., Spry, S., and Hedrick, J. K. (2005). Optimal path planning in a constant wind with a bounded turning rate. In *AIAA Guidance, Navigation, and Control Conference and Exhibit*, pages 1–11.

[McKay et al., 2000] McKay, M., Beckman, R., and Conover, W. (2000). A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61.

[Pepper, 2012] Pepper, T. (2012). Drones–ethical considerations and medical implications. *Journal of the Royal Naval Medical Service*, 98(1):37.

[Raymer, 2006] Raymer, D. P. (2006). *Aircraft Design: Aconceptual Approach Fourth Edition*. American Institute of Aeronautics and Astronautics.

[Sarris and ATLAS, 2001] Sarris, Z. and ATLAS, S. (2001). Survey of uav applications in civil markets (june 2001). In *The 9 th IEEE Mediterranean Conference on Control and Automation (MED'01)*.

[Savla et al., 2005a] Savla, K., Bullo, F., and Frazzoli, E. (2005a). On traveling salesperson problems for dubins vehicle: stochastic and dynamic environments. In *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pages 4530–4535. IEEE.

[Savla et al., 2005b] Savla, K., Frazzoli, E., and Bullo, F. (2005b). On the point-to-point and traveling salesperson problems for dubins' vehicle. In *American Control Conference, 2005. Proceedings of the 2005*, pages 786–791. IEEE.

[Techy and Woolsey, 2009] Techy, L. and Woolsey, C. A. (2009). Minimum-time path planning for unmanned aerial vehicles in steady uniform winds. *Journal of guidance, control, and dynamics*, 32(6):1736–1746.

[Teixeira et al., 2008] Teixeira, J., Stevens, B., Bretherton, C., Cederwall, R., Klein, S., Lundquist, J., Doyle, J., Golaz, J., Holtslag, A., Randall, D., et al. (2008). Parameterization of the atmospheric boundary layer: a view from just above the inversion. *Bulletin of the American Meteorological Society*, 89(4):453–458.

[van Blyenburgh, 2000] van Blyenburgh, P. (2000). Uavs-current situation and considerations for the way forward. Technical report, DTIC Document.

[Weibel, 2005] Weibel, R. E. (2005). *Safety considerations for operation of different classes of unmanned aerial vehicles in the national airspace system.* PhD thesis, Massachusetts Institute of Technology.