Autonomous Underwater Vehicles (AUVs) have the potential to replace or augment existing manned or remotely piloted underwater vehicles in the completion of hazardous and tedious tasks. However to do so, a robust method of navigation and collision avoidance is required before these vehicles can operate fully autonomously. This thesis determined from the state of art of path planning, that the randomized A* search algorithm (R*) developed by Anthony Stenz and Maxim Likhachev was suitable for modification to better fit the unique requirements of the AUV being developed at ADFA. The algorithm was modified to incorporate a kinodynamic successor node generator, that can be described as a combination of the motion primitives method and a heuristic dynamic model based planner method. The path planner was then tested against multiple obstacle maps to test its robustness and optimization. The path planner is able to find a path where one exists however the increased heuristics and integration of kinodynamics significantly increased the required computational effort to calculate a path. This however is rectifiable with smarter programming and is worth further development as the path planner is designed with an operational vehicle in mind.

Contents

I. Introduction 3
   A. Background 3
   B. Aims 3
   C. Path Planning Requirements 3
   D. Path planning Research Summary and Algorithm Selection 3
   E. R* theory 4
   F. Kinodynamic Approach Selection for Incorperation into the R* Algorithm 5

II. Path Planning 2
   A. Deconstruction and study of similar path planning algorithms implemented in Matlab 5
   B. Heuristics in R* Sub Program 5
   C. Heuristics in wA* sub program 7

III. Kinodynamics 9
   A. Approximation of Vehicle Dynamics 9
   B. Linear and rotational velocity calculations 10
   C. Linear and rotational acceleration as a function of control inputs 10
   D. Generating the motion primatives 13

IV. Conclusions 14

V. Recommendations 14
   Acknowledgements 14
   References 14

APPENDICES

1 PLTOFF, School of Engineering & Information Technology. ZEIT4500/4501/4297.
Nomenclature

Terms:
- **AUV** = Autonomous Underwater Vehicles
- **SLAM** = Simultaneous Localisation and Mapping
- **node** = Graph theory representation of spatial coordinates.
- **Cnode** = The currently evaluated node in the path planner not the actual vehicle state prior to planning
- **Snode** = Successor node (Node to be explored as chosen by lowest k value)
- **Tnode** = Target node (End point of short range path)
- **Gnode** = Goal node (End point of total R* path or Grand Path)
- **OPEN** = List of nodes sorted by k value to obtain next Snode
- **CLOSED** = List of all Snodes fully explored, used for backtracking to record path computed.
- **GPS** = Global Positioning System
- **DOF** = Degrees of Freedom
- **AVOID** = Identifier for determining whether a path is successful, stopped by the expansion limit or has no solution

Variables:
- **dt** = time step [s]
- **i** = state
- **θ** = orientation [rad]
- **u** = AUV forward velocity [m/s]
- **r** = rotational velocity [degrees/s]
- **δr** = rudder deflection [degrees]
- **RPS** = rotations per second
- **h** = estimated cost between nodes
- **g** = path cost so far
- **k** = priority value
- **x** = xpos = global x-position [m]
- **y** = ypos = global y-position [m]
I. Introduction

A. Background

Autonomous underwater vehicles have the potential to fulfill numerous underwater roles that are currently restricted by size, cost and the limitations of manned support. Gwyn Griffiths outlines future roles AUVs might fulfill that are otherwise restricted to remotely piloted or manned underwater vehicles and the benefits autonomy might bring. The benefits removing manned operators include reduced cost, increased safety and increased performance in the underwater environment. Autonomy also means that roles previously dismissed as being too restrictive or hazardous for manned systems can now be explored. Wadhams and Wilkinson, for example, explore a potential role for AUVs in tracking the flow of oil under sea ice, to better model and combat oil spills in ice covered seas.

B. Aims

This thesis follows closely an existing UNSW@ADFA project by Kirsanov Andrey to develop an AUV system however; this thesis focuses only on developing a path planning algorithm for the AUV. The path planning algorithm must be suitable for use with only a partial knowledge of the environment and deal with time variant changes in obstacles. The algorithm must also factor in the kinodynamic constraints of the AUV when generating a trajectory to be followed.

A. Path planning Requirements

The unique requirements of a path-finding problem determine the path planning approach. Therefore the path-planning problem was first simplified into various path-planning requirements that are likely to be essential in the AUV environment and was used as a basis of comparison between algorithmic approaches.

The research was constrained to offline or global path-planners that calculate the entire path before the first movement. This was due to the low AUV velocities and their resistance to dead ends unlike online bug-type algorithms. The AUV must periodically surface to correct inertial drift using a GPS position fix. This provides an opportunity to calculate the entire path and is another offline planning advantage. This however adds the requirement for quick computation of paths. The path-planning algorithm chosen must also be able deal with partially known environments and be able to re-plan to accommodate changes. This is because the knowledge of an environment is often incomplete and will change as the vehicle progresses through it. This is particularly so with the underwater environment as only an estimated 5% of the oceans seafloor has been mapped in detail.

The path-planner must in addition take into account the trajectory of the moving obstacles, as real vehicles do not have infinite velocity and instantaneous acceleration needed to assume moving obstacles are static. The configuration space or C-space represents all possible states of the vehicle and is an extension of the obstacle map to account for the DOF of the vehicle and time. Using a path planner that does not require a C-space with a large number of dimensions is important as computational constraints such as memory size and processing power dictate the maximum amount of information a representation of an environment can convey. The AUV’s size and power requirements dictate these computational constraints. Finally, the AUV has constraints on velocity, acceleration and turning radius. They will also be a non-spherical shape with significant mass. Kinodynamic planning refers to incorporating the vehicles shape and orientation constraints into path planning. Kinodynamic planning extends kinematic planning by incorporating a vehicle’s dynamic constraints such as velocity, acceleration and inertia. Therefore the path-planner must be able to integrate kinodynamic planning to be useful for a real vehicle.

These requirements of an offline planner that can deal with partially known environments, moving obstacles and incorporates kinodynamics without an impractically large C-space form the basis of selection of the path planning algorithm for modification.

B. Path Planning Research Summary and Algorithm Selection

A broad review was conducted of the developments in path planning from 1970 up to 2013. The requirements detailed in the previous section were used as the basis for evaluation. Mashehian’s and Sedighizadeh’s chronological review of motion planning over 35 years was a helpful starting point. Path planning algorithms can be described as classical or intelligent. The difference being intelligent algorithms use meta-heuristics, changing conditional rules, to optimize an initial solution. Classical algorithms in comparison only use heuristics, fixed conditional rules, to restrict the application of the base algorithm. The practical implications are that classical methods are guaranteed to find the optimal solution whereas intelligent algorithms will find a solution extremely quickly, but may be excessively far from optimal.
The majority of papers separate the classical algorithms into three main approaches.\textsuperscript{13} Cell decomposition approach, potential field approach and roadmap approach being the three most common classical approaches.\textsuperscript{14} Each method has advantageous and disadvantageous and the differences relate to how the algorithm interprets the environment. A brief comparison of the three main approaches is included in Annex A. Masehian, E., and Sedighizadeh, D., in their survey of 1381 robot motion planning papers,\textsuperscript{15} indicate that intelligent algorithms such as genetic algorithms, artificial neural networks and potential field hybrid methods are the most popular avenues of new research between 1998 to 2007. This may indicate that research into classical algorithms has maturated and intelligent algorithms offer the most promising avenue to develop path planning further.

The purposes of this thesis, however, are to adapt a path planning algorithm to the AUV being developed at ADFA. Therefore a classical path planning approach is the safer choice for an operational vehicle as current research into intelligent algorithms still focuses on robustness.\textsuperscript{16} This will unfortunately be at the expense of accuracy, as computational constraints will restrict the number of the vehicle’s DOF that can be considered when choosing a path. The benefit to choosing a classical algorithm is over 40 years of research into this area, ensuring a large pool of knowledge from which to learn and implement the path planning method. In choosing between classical methods, kinodynamic constraints must be considered to be as critical to path planning as obstacle avoidance. RRT and A* derivatives like D* lite offer the advantages of being well researched and applied and meet all the criteria mentioned in Section A. Path Planning Requirements. RRT has the benefit of integrated kinodynamic constraints and computational speed whereas A* derivatives offer the advantage of flexible path evaluation criteria and more complete optimality. Randomised A* (R*) is an algorithm that combines these advantageous, of A* derivatives and RRT, with the additional advantage of strong performance in the presence of large local minima.\textsuperscript{17} Local minima are concave obstacles that are a particular problem to classical path planning methods.

The R* algorithm was developed by Anthony Stenz and Maxim Likhachev\textsuperscript{17} and is the base algorithm selected.

C. R* Theory

The R* algorithm functions like the RRT method at the macro level generating random target nodes at a set distance from the end of a branch. However instead of kinodynamic or heuristic constraints generating the branches, the A* derivative wA* calculates the branch path. The wA*path planning algorithm is based on graph theory.

Graph theory is a study of mathematical structures used to model relationships between objects. Typically a graph is made up of nodes and the connections between nodes called edges. In path planning it is common to directly relate nodes to spatial coordinates and the edges as some real world cost (h) to get from one adjacent node to another such as Euclidean distance see Eq. 1. Possible paths are found by following adjacent nodes to the target node and the total path cost equals the sum of the edges of the path.\textsuperscript{18}

\[
h_{(n,n+1)} = \text{Euclidean distance} = \sqrt{(xpos_{n+1} - xpos_n)^2 + (ypos_{n+1} - ypos_n)^2} \tag{1}
\]

Exploration of a node involves searching the adjacent nodes around it. Every searched node is placed in a list where the next node to be explored is selected from. This is called the OPEN list. The wA* search algorithm assigns a cost to each searched node in terms of a sum of the distance between all the adjacent nodes in the path so far, Eq. 2, and a straight line estimate of the cost to go, the distance from current node to target node, Eq.4.
The R* planner only allows the wA* a limited number of expansions to reach the target node. The paths, generated to the target nodes, are then ranked based on a priority value calculated exactly the same as the one used in the micro wA* search, Eq. 3, to evaluate the paths. Additional, paths that did not complete due to the expansion limit are ranked lower than those that did. The best path’s end point is chosen for the next start point of the random target generation. Figure 1 shows the reduction in expansions and hence computational time R* has compared to wA*.

D. Kinodynamic Approach Selection for Incorporation into the R* Algorithm

There are three methods of incorporating kinodynamics into R*. The first is to utilise a trajectory planning function to smooth the existing non-kinodynamic constrained path using, for example, the B-spline function. This is commonly used in UAVs due to its computational speed and the relatively sparse obstacle nature of the environment. The second method is to hard encode heuristic constraints into R* so that violations of kinodynamic constraints are discarded. The third method and the method chosen is the use of motion primitives to restrict the possible movements at each state to a few pre-computed paths that comply with the constraints. This approach without modification is restricted to only a few speeds and insufficiently accounts for vehicle acceleration and rotational velocity. Thus the modification perused in this thesis is to create continuously adaptive motion primitives that take into account the vehicles current forward and rotational velocity and acceleration and integrate them into the R* path planning approach.

It must be noted that application of simple motion constraints and moving obstacles to the R* algorithm have been implemented and the author of the R* algorithm Maxis Likhachev has already indicated a desire to fully implement dynamics into the algorithm in 2010. However as of May 2013 the literature survey on R* conducted did not find an implementation of the R* algorithm that involves kinodynamic constraints requiring the high dimensionality of varying accelerations, velocities, holonomic constraints as well as moving obstacles and consideration for inertia required of realistic AUV path planning in one algorithm.

II. Path Planning

The aim of this projects is to integrate kinodynamics into the R* algorithm. The R* algorithm uses heuristics to evaluate, sort and choose the paths created by the wA* sub program, to make up the grand path to the goal node. The wA* sub program itself evaluates the successor nodes (Snodes) that are generated by the kinodynamic Snodes generator.

A. Deconstruction and study of similar path planning algorithms implemented in Matlab

The two Matlab codes studied that contributed the most were the A* implementation one developed by Paul Premakumar, Fig. 20 Appendix B, and the second created by Drexel Autonomous Systems Lab at Drexel University. These were useful in developing the basic rules and structure of the wA* planner section of the R* Matlab code. These codes however were unsuited for large maps due to the use of searching functions in examining the CLOSED and OPEN lists for obstacles, and could not be directly modified to build the R* planner due also to the need to integrate the kinodynamic Snodes generator.

B. Heuristics in R* sub program

A flow chart of the interaction of the Heuristics in the R* sub program is shown in Appendix C

The map of a known environment is almost always a picture file, thus it is appropriate for the path planner to input the map as a picture file. Map input is via black and white Portable Network Graphics File (.png) instead of the more common method of putting obstacle coordinates on the CLOSED list. This has the advantage of allowing navigation through unstructured and complex obstacle environments, without the need to define the boundaries of an obstacle. The same is true for a robotic system in a truly unknown environment as sensor outputs from Sonar, Radar, LIDAR, visual and other sensors all need to be run through a simultaneous

Figure 1 Nodes expanded are shaded grey. Comparison of A*, wA* and R* in circumnavigating a large local minima

---

---
localization and mapping (SLAM) program. These programs will also generate a pictorial representation of the environment to be displayed to humans which can be inputted into the path planner. In both cases changing the resolution and file output types are relatively straightforward tasks. Each pixel is used as the cell in the cell decomposition of the map and changing the resolution defines how close the AUV can navigate to obstacles.

The pictures are run through a simple function that increases the boundaries of every obstacle to account for the dimensions of the vehicle. Picture files have an axis system where the positive direction of y is down and thus all coordinates used in the path planner reflect this. Thus the Start and Goal nodes (Gnode) are expressed in terms of this axis. This is complicated by Matlab’s method of calling matrix coordinates which based on rows and columns where the y, or row, coordinate is called first. Due to the integration of kinodynamics the starting node must also specify a starting forward velocity, orientation and rotational velocity. Whilst it is a simple method to specify these additional parameters for the goal node, a significant amount computational effort is involved in simply reaching near the coordinates. Specifying more goal parameters other than forward velocity must be zero was deemed to be unnecessary. Even this parameter is only accounted for by using a separate braking function that runs the kinodynamic Snode generator in an iterative loop until the forward velocity is zero.

The path planner is also designed to work with any size map, thus the boundaries of the map picture. If the Gnode is close to the start node then there is little reason to generate multiple short range paths in random directions to reach it. Therefore a heuristic switches from R* to wA* as the end point or the start node approaches within 200m straight line distance. If the path planner cannot find a path within the expansion limit then it will not try from the end point of a random generated path but rather from the end point of the latest wA* attempt. If there is sufficient distance between the start node or end point of the latest best short range path then one to eight random targets nodes for short range paths are generated. To eliminate the possibility of the path being generated from the same start and end point the random points are selected from a pool.

The random target pool was generated with the separate circle function. The coordinates of which are rounded to integers. Increasing the length of the short range paths by increasing the radius of the circles results in more expansions needed to calculate the path, hence a longer computation time per path. As the radius decreases below 100m the inner circle disappears. The smaller radius takes a shorter time to compute each path but results in less progress per set of random target nodes. The radius increases if all paths in a set are successful and decreases if no paths are successful. Unfortunately the greater the radius the more likely the target nodes are on the other side of a large obstacle that is impassable within a reasonable number of expansions. This reasonable number of expansions was observed to be around five times the radius. The net effect observed is that the greater the radius, the shorter the total computation time, as long as there are few obstacles. Thus in areas cluttered with obstacles the path planner should use a shorter radius in the random target selection pool and increase in radius as more free space is available.
The random nodes selected from the pool are transposed with reference to the start node of the path. A check is also performed to ensure all chosen nodes are within the boundaries. Any random nodes that intersect with obstacles are left for the wA* planner to conduct an obstacle check. As using a heuristic to ensure all target nodes are not obstacles resulted in the bunching of multiple short range paths that traversed the same general area especially in narrow corridors.

To ensure the path planner does not get stuck in shifting between the best two or more path end points that always result in obstacles as their target nodes, think multiple dead ends, an additional heuristics is applied. This heuristic searches the entire list of previous path start and end point combinations dozens of times and thus unfortunately slows the program down as the path planning progressed and the number of path combinations increased into the thousands. A solution would be to search only the nearest dozen or so paths however in one case a logic loop between four dead ends, hundreds of meters apart, was encountered thus to create a robust system the entire list is still searched.

The start and target nodes are then exported to the wA* planner, research has been conducted proving that the wA* planner in the R* algorithm can be integrated with parallel computing. The code reflects this by grouping the random target nodes together inside a separate ‘for’ loop instead of sequentially with the random target generation so that the Matlab function ‘parfor’ could be integrated later. Grouping allows paths to be completed out of order, a requirement of parallel computing. Parallel computing whilst accounted for, is outside the scope of this research.

The paths created by the wA* planner for each random target node combination are saved in a separate list that does not get sorted. This is because the list contains tens of thousands of entries and continuously sorting and searching the list every eight paths is an unnecessary waste of computational effort. Instead the location of the starting and ending nodes of the paths are recorded as an entry into a separate list that contains the identifiers for all the tried combinations. This list also contains the path costs, and all the kinodynamic details generated and needed by the wA* planner. A ‘parent set identifying number’ is included to show the path is a descendant of the most recent set of random targets. This list of path identifiers is sorted first by whether a path is complete, stopped by the expansion limit or is a failed path search. In this way all easily achievable paths will be explored before the harder paths are chosen. It is then sorted by priority value with the path endpoint of lowest priority value being chosen to be the origin of the next set of short range paths. The wA* planner sub program uses a different cost function to that normally used and used in evaluating the paths. The cost function used in the R* at the macro level is the original wA* cost function used in the literature, Eq. 2, Eq. 3 and Eq. 4.

After the best path’s end point is selected to be the next start point, the path identifier is transferred to a separate list that will not be sorted. This isolates the node to ensure that the path is not recomputed, it also allows the path identifiers to be tracked back to the start node to assemble a grand path from start to goal. This back tracking function uses the parent set identifier along with the path start and end point identifiers for each path segment to assemble a grand path. This back tracking function only initiates after the main path planning program terminates.

There are four reasons that the path planner terminates, the first is if the end point of one of the short range paths intersects with the goal node. The second is if the end point is within a set distance from the goal node, this is to reduce the amount of unnecessary maneuvers, looping, close to the goal node. This is set to 10 meters. The third is if there is no possible path to the goal node. The fourth is a limit on the total number of sets of short range paths that can be tried, this is set to 1000 or a maximum of 8000 path assessed. This is important since the maximum number of paths that could be generated in a one kilometer squared map of 1 meter resolution, or 1 million nodes, approaches infinity.

C. Heuristics in wA* sub program

Annex D contains the flow chart of the interactions between the heuristics of the wA* sub program.

A boundary and obstacle check is preformed on the selected target node to avoid searching for a path guaranteed to have no solution. To avoid progressively slowing the path planning the wA* planner rinses the CLOSED and OPEN list. Keeping the lists would result in better paths by sharing previously searched paths progressively as each run adds hundreds of entries into the CLOSED and OPEN the time taken to systematically search them in the later stages of the path computation, would increase dramatically.

The core of the wA* planner is the evaluation of the Snodes that are generated by the Kinodynamic section of the R* planner. The kinodynamic planner takes the full Current node (Cnode) information including its dynamic state and generates a list Snodes to be evaluated. These Snodes reference the Cnode using a parent ID,
to be used in later path assembly. Each Snodes is evaluated one at a time. The Snodes is tested to ensure it is within the boundaries of the map. An obstacle check is performed on the planner by calling the coordinates on the picture map and testing the pixel to see if it represents an obstacle or free space. This method of testing for obstacles is common for all functions in the R* path planner. Two checks are performed to see if the Snodes has already been explored, if so there is no need continue evaluation as the point already has Snodes of its own. However if the Snode is referenced by another path, but Snodes have not been generated from it, i.e. not explored, then the two parent paths are compared and the better path is referenced as the parent. This prevents the recalculation of the same route whilst still allowing paths to get progressively better as more nodes are explored.

Once the Snodes has been vetted as being unique and belonging to the best hereditary path, the cost can be calculated for each Snodes. The ‘g’ value or cost so far is calculated by the kinodynamic Snodes generator and reflects the ranking of energy cost involved in the maneuver to reach the Snodes. This is necessarily different to the method used in calculating cost so far, when evaluating complete paths, as the Snodes do not have data on the points between the Cnode and the Snodes. Thus the distance traveled could not take into account the curvature of the path when comparing similar Snodes. Energy cost can also correctly rank a Snodes that uses far less energy but travels a slightly further distance than a maneuver that applies tremendous force to overcome inertia to shorten the path by an insignificant distance. The estimated cost to go is the straight line distance between the Snodes and the target node. The priority ‘k’ value is calculated using Eq. 3 where the weight ‘w’ must reflect the difference in measurement between an arbitrary energy cost ranking and a distance. The value for weight used is 20 and was based by qualitative analysis of the generated paths. The greater the weight the more preference for reaching the target rather than taking excessively long times to reach the target by costing at minimum RPS. Too much weight however and the path calculated ends up including detours in dead ends, due to the minimum ‘h’ value of Snodes within those detours and the low importance of path length. An optimization study of the value of ‘w’ would have to take into account the type of obstacles to be encountered. With higher weights perfect for maps without many obstacles.

The vetted nodes, now assigned a cost, populate the OPEN list. The next Cnode is selected from this list which is sorted by the lowest priority value after all the Snodes of the current Cnode have been evaluated. The Cnode corresponding to the lowest priority value, represents the Snodes that results in best ‘progression to the target’ vs ‘energy cost ranking’ balance dictated by the w value. This new Cnode is transferred to the CLOSED list as it is fed into the kinodynamic Snodes generator and hence will be fully explored.

The kinodynamic Snodes generator has an input for the time increment it should use for Snodes generation. Basically the greater the time increment the more ground the Snodes cover, the faster the path planner completes. Due to the nature of the kinodynamic generator the accuracy of the Snodes state to that of the real life AUV will reduce with increasing time increment.

Larger time increments may also result in paths that jump thin obstacles. A possible fix is to run the full completed R* path once more through the kinodynamic generator at a smaller time step, and retest for obstacle intersection. If any intersections are found then the offending short range path is rerun through the wA* planner at the minimum time interval. A minimum interval is necessary as decreasing time step too much will mean that the generated Snodes positions will move a distance less than the resolution of the Cell decomposition. This will result in the planner listing nodes as fully explored with all the Snodes being the same node and hence the planner will finish with a ‘no paths available’ termination condition since all maneuvers have been explored with no result.

The exhaustion of the OPEN list corresponds to the exploration of every possible Snodes and the wA* search terminates as a failed search. This can only occur if no path is available to the target node. However if the Cnode selected has the same coordinates as the target node then the search also terminates. It was found that with this termination condition the path often overshoots the target node then doubles back, especially with large time steps, As shown in Figure 5 Paths double back and loop due to strict termination conditions below.
This was fixed by relaxing the termination conditions so that the path was deemed complete if the current node was within 100m. This relaxation is of no consequence as the targets are randomly generated anyway, and the resulting end point is orientated in a more preferable direction.

The strength of the R* algorithm lays in avoiding the exploration of difficult paths and thus an expansion limit is introduced. The minimum number of expansions, or nodes explored, occurs in a straight-line path with no obstacles in-between. The more convoluted the path the more expansions. Since exploring nodes takes time the expansion limit ensures that the wa* path does not spend too much time exploring possible paths. It also removes the need for spatial boundaries to the wa* sub program.

The backtracking function used to generate these paths are the same as the one used in the R* main program to generate the grand path. The difference is that the parent identifiers relate nodes not paths in the CLOSED list. Another difference is that while the R* will keep searching until a path is found, the backtracker on the wa* sub program, must differentiate between failed paths, expansion limit paths and successful paths. Paths that have either got within the specified termination distance or hit the expansion limit, are exported to the main R* program. The latter is assigned an AVOID identifier of 1 so that the R* planner will only use the end point of the half finished path, as a new start node for the Random generation, only if all successful paths have been explored. The successful paths are assigned an AVOID identifier of 0 and the failed searches assigned an identifier of 2, thus making it easy for the R* program to sort the path identifiers using the AVOID values before sorting path priority values.

III. **Kinodynamics**

To ensure that the path generated by the path planner complies with the vehicles kinodynamic constraints, the constraints are integrated into the path planner via continuously adaptive motion primitives. These motion primitives are generated based on the state of the vehicle at the current node. The motion primitives are created using mathematical logic that approximates the dynamics of the vehicle. The mathematical logic is the inverse dynamics of the vehicle generated from the responses of the vehicle to control inputs.

**A. Approximation of the vehicles dynamics.**

The inverse dynamics of the vehicle is based on a simplification of the dynamics of the AUV best described as that of a car in reverse. This simplification is appropriate as the AUV, similar to a car, has only two controllable DOF, forward motion and rotation, yet can move in 3 DOF, x, y and orientation. This system is described as non-holonomic since the controllable DOF are less than the total DOF. Modifications were made to the non-holonomic car dynamics to resolve the dynamics as functions of forward velocity ‘u’, acceleration, orientation, rotational velocity ‘r’, rotational acceleration and position. This was done as it is easier to measure the linear and rotational acceleration and velocity due to a control input than to measure force in an operational AUV.
These equations of motion are a linear approximation of a non-linear motion thus can only used for small time increments. Whilst a more valid model could be used these equations are run for the 88 different control input combinations for each Cnode, thus the accuracy of the approximation of the dynamics is sufficient given the potential loss in computational speed of a more valid alternative. These equations of motion are an operation in the MATLAB file dyn_select.m which is a function of the main R* path planning code and is attached in Appendix B. The 88 combinations involve 8 thrust settings and 11 rudder deflections.

B. Linear and rotational velocity calculations

By examining the vehicle’s dynamics it was found that the vehicle’s rotational velocity can be decoupled from the thrust setting of the vehicle, since the rudder is forward of the thruster. This allows the vehicle’s forward acceleration to be approximated quite accurately by separating the forward acceleration as a function of forward velocity for various thrust settings only and that for various rudder settings only.

First the vehicle’s current forward acceleration is calculated from its current forward velocity and thrust setting, then the forward acceleration is adjusted to account for the drag due to rudder deflection.

The same process is applied to the rotational acceleration whereby the rotational acceleration is separated into the components due to rudder deflection and that due to drag as functions of forward velocity and rotational velocity respectively.

The forward and rotational velocities can now be calculated from the equations of linear acceleration.
The velocities used in the vehicle dynamic’s equations of motion are the average between the final and initial linear and rotational velocities to account for the change in velocity over the time step. These operations are conducted in the MATLAB file dyn_select.m which is a function of the main R* path planning code and is attached in Appendix B.

C. Velocity and rotational acceleration as a function of the control inputs

The equations of motion described in the preceding two sections require the relationship between vehicles control inputs and its responses in order to create the motion primitives for all possible maneuvers given a vehicle state. This involves creating the relations Eq. 8, Eq. 9, Eq. 11 and Eq. 12. The AUV used in this project has had its dynamics modeled in SIMULINK for controller design for the AUV as part of a Doctoral Thesis by Osama Hassan. The SIMULINK model had to be slightly modified to accept control inputs directly instead of a path that a controller is required to follow. It was also modified to display and record data on the linear and rotational acceleration and velocity.

Using this model the data was manually generated by first plotting the acceleration vs. time and velocity vs. time plots for the acceleration due to eight different thrust settings (RPS = 0, 1, 2, 3, 4, 5, 6, -4) with a starting velocity of -1.049 m/s (The maximum reverse velocity at a thrust setting of RPS = 4) with no deflection. This was repeated with a starting velocity of 1.407 m/s (The maximum forward velocity at a thrust setting of RPS = 6). Then the two plots assembled into a continuous plot of acceleration as a function of forward velocity of -1.049 to 1.407 m/s. The curve function of this plot forms the relation Eq. 8 and is shown in Fig. 7 below.

Then the velocity, acceleration and rotation rate vs time was plotted for the deceleration due to rudder deflections of 0 to 25 degrees in 5 degree increments. The deceleration curve for a thrust setting of zero and rudder deflection of zero was then subtracted from the plots to isolate the deceleration due to rudder deflection only. This is in accordance with Eq. 13. The deceleration was measured from the maximum forward and reverse velocities, 1.407 m/s and -1.049 m/s respectively. The curve fitted to this is the relation Eq. 11 and is shown in Fig. 8.

Rotational acceleration as a function of velocity was measured for each rudder deflection for forward velocities of 0 to 1.5 m/s in 0.1 m/s increments. This was measured at the first SIMULINK time step which corresponds to a rotational velocity of zero and thus corresponds to the rotational acceleration without the influence due to drag. The curve fitted to this is the relation Eq. 11 and is shown in Fig. 9.
The rotational acceleration due to drag is a function of rotational velocity. Thus a plot is created measuring the rotational acceleration and velocity whereby the rotational velocity decelerates from a starting rotational velocity of 4.88 degrees per second with a thrust setting of RPS = 0, a rudder deflection of zero. This starting velocity was found to be the maximum rotational velocity and resulted from a thrust setting of RPS = 6 and a rudder deflection of 25 degrees. The curve fitted to this is the relation \( \text{Eq. 12} \) and is shown in Fig. 10.

Determining the rotational acceleration due to drag as a function of rotational velocity may be difficult to do experimentally. An alternative explored with similar results was to first generate both the total rotational acceleration for all rudder deflections through the full range of forward velocities which is a function of both forward velocity and rotation rate. Then subtract the drag relation \( \text{Eq. 12} \) due to rotational velocity to get \( \text{Eq. 11} \). This however results in an error that must be addressed due to the change in angle of attack of the rudder deflection; see Fig. 11, caused by perpendicular flow across the rudder which in turn is caused by the rotational velocity induced by the rotational acceleration due to rudder deflection.

Alternatively the lift force equation for the rudders may be used to find the drag relation from the total rotational acceleration instead. These three methods are simply operations based on \( \text{Eq. 13} \) which is illustrated in terms of forces in Fig. 12 above. The reason for the splitting of the components due to drag and that due to rudder deflection is so the rotational acceleration is correctly dependant on the direction of rotation (drag resisting rotation) of the AUV at the Cnode and not just forward velocity and control inputs.

The operations to generate the relations \( \text{Eq. 8, Eq. 9, Eq. 11 and Eq. 12} \) are valid because the rotational and linear accelerations and velocities are all measured on the same time step for each SIMULINK run and that would be true for an experimental run as well. These operations are conducted in the MATLAB file vel_incr.m in Appendix B. This MATLAB file is not a main function and can be either run before or parallel to the main R* path planning algorithm. An advantage of running in parallel in an operational AUV scenario is to continuously update the dynamics based on accelerometer readings whilst the AUV is traversing the path. This may result in more robust and a potentially more accurate dynamic model of the vehicle. The use of a polynomial curve fitting function in vel_incr.m reduces the effect of outliers making this more feasible. Appendix C describes a self calibration path that can be run prior to the path planner being utilised in a new aquatic environment, or after changes to the AUV’s configuration.
D. Generating the motion primitives

At every Cnode the current forward and rotational velocity is used to calculate the acceleration and velocities using relations Eq. 8, Eq. 9, Eq. 11 and Eq. 12 for each of the 88 combinations. These accelerations and velocities are then used to calculate the velocities used in the simplified dynamics of the AUV. The calculations involving the simplified dynamics of the AUV result in 88 unique positions coordinates and orientations, an example is shown in Fig. 13.

The AUV dynamics are calculated in a body-axis reference frame and the easiest method transferring the body-axis coordinates to a global reference frame is to utilize Euler angle rotation matrix below after normal translation, Eq. 16 an example of which is shown in Fig. 15.

\[
R = \begin{bmatrix}
\cos \theta & -\sin \theta \\
\sin \theta & \cos \theta
\end{bmatrix}
\]  

(16)

Since the map resolution being used is one metre squared these positions need to be rounded to the nearest metre, shown in Fig. 15. The smallest deflection, lowest RPS, control combination is chosen out of control combinations with the same rounded coordinates. These coordinates are the Snodes used in the wA* section of the base R* path planning algorithm. The cost function evaluating the Snodes can no longer be a simple Euclidean distance equation since this does not take into account the curve of the path. Therefore a simple ranking of the energy cost of the control inputs is used whereby the greatest rudder deflection at a thrust setting uses less energy than a control combination of the next thrust setting higher with zero rudder deflection.

IV. Results

The performance of a path planner is dependent on the complexity and size of the obstacle map. These are difficult to quantify therefore a qualitative analysis of the path planner performance is conducted. Firstly, the kinodynamic Snode generator is compared to the SIMULINK dynamic model of the vehicle. The kinodynamic wA* planner is then run against several paths that present difficulties to other path planning approaches. Then the R* planner with kinodynamics is compared to an R* planner without kinodynamics.

The kindynamic planner was tested for one, three, six and nine second time increments. The results shown in Fig. 21 Appendix F, indicate that as the time increment increases so does the error between the SIMULINK dynamic model and the results from the Kinodynamic Snode Generation function, dyn_select.m. Details of the two methods of determining the vehicle responses, shown in Table 1 Appendix F, reveal that the position error for the one and three second time increment is less than 5 mm. While the error in the forward and rotational velocity at the three second interval is less than 0.03 m/s and 0.09 degrees/s respectively. This validates the accuracy of the kinodynamic Snode generator against its purpose as a short time increment approximation of the vehicle dynamics. The figures in Table 2 Appendix F, show that despite a complex response trajectory the kinodynamic Snode generator maintains positional accuracy. Time intervals of six, nine and greater are not practical as it would severely restrict the AUV if it had to hold a control input for no less than that time increment thus the loss of accuracy is inconsequential.

The wA* path planner coupled with the Kinodynamic Snode generator are tested against obstacle maps that challenge other path planning approaches. The resulting paths shown in Table 3 Appendix G indicate that the kinodynamic Snode generator does not remove the advantages of the cell decomposition method over the other
path planning approaches. The yellow shading on the maps in the left column indicate regions of explored nodes and red points on the maps of the right column indicate the positions of the AUV separated by time increments of 6 seconds along the final trajectory.

The R* path planner without kinodynamics generates paths with sharp turns that require a controller with collision avoidance and short range path planning to follow. Figure 37 Appendix G shows the sharp 135 degree change in direction and the 90 degree turns along the sides of the obstacles that require a controller to pre-empt the maneuver to avoid collision from overshooting caused by vehicle dynamics. The R* with kinodynamics generates paths that are possible to follow because every Snod is generated based on the dynamic model of the AUV. Thus the AUV slows down (distance between points indicating time increments reduce) before making sharp turns and turns are made over dozens of meters. This is shown in Fig. 38 Appendix G.

V. Conclusions

The R* path planner modified to incorporate kinodynamics was able to compute very long paths in complex terrain whilst complying with kinodynamic constraints. This indicated by paths that were generated without intersecting obstacles or attempting impossible position changes. The trajectory generated shows smooth curves that require the vehicle to slow down for sharp turns. The building blocks of the path, the motion primitives were proven to match the dynamic model of the vehicle for small time increments. Thus the aim of the thesis of combining kinodynamic constraints with the R* search algorithm was achieved.

VI. Recommendations

Incorporate parallel computing as even mobile phone processors are multicored now. Remove the need to perform searching functions through list as this will significantly reduce computational exosts of the path planner. Perform an optimisation study on the numerous parameters involved. Develop a dynamic model that is less computationally costly but better fits the data. Integrate the path planner into a real AUV.

Acknowledgements

I would like to thank my supervisor Dr Sreenatha Anavatti for his excellent feedback, support and mentorship during the completion of my thesis. I would also like to thank Osama Hassan for his help in understanding and working with his SIMULINK AUV controller. I also thank Dr Murat Tahtali for his help with programming in Matlab.

References

10 Goodwin, S.D., Menon, S., Price, R.G., “Pathfinding in Open Terrain,” Centre for Interactive Digital Entertainment Research, School of Computer Science, University of Windsor, Windsor, Ontario, Canada, 2006
Appendix A: Brief comparison of Classical Path Planning Algorithms

The cell decomposition method discretises the configuration space into a set of cells. A cost function assigns a cost to each cell and the path planning problem becomes determining the continuous sequence of adjacent cells from the start cell to target cell with the lowest total path cost. The A*, D* and D* lite methods are effective and well-researched methods of determining this continuous cell sequence. Approximate cell decomposition methods such as grid or quadtree, use regular shaped cells most often squares, cubes in 3D, to represent the environment. Accuracy is inversely proportional to the size of the grid cells resulting in high computational costs which is the main driver for alternate path finding methods.

The potential field method, assigns a resistive potential to every obstacle simulating a resistive force on the vehicle. The target position is assigned the lowest potential with the potential increasing continuously with increasing distance from the obstacle. The path-planning algorithm then seeks a path that will cause the vehicle to move from the highest resistive potential to the lowest. The potential field approach has the considerable advantage of being computationally simple and can be run as an online algorithm. This method’s major fault lies in the ability for vehicle to become stuck at local minima whereby no further action by the vehicle will result in an immediate decrease of the potential.

The roadmap method does not compute the entire configuration space but rather a discrete set of connected nodes or paths, significantly reducing the computational cost. The two most common methods are the vertex and voronoi, path planning approaches. These approaches suffer from the too-close or too-far from obstacles problem where the path deviates away from the optimal path because the approaches use the obstacles to calculate waypoints. The Rapidly Expanding Tree (RRT) roadmap approach, generates paths that sequentially and randomly branch out in C-space according to a heuristic. Eventually terminating once the target node is connected is reached. Heuristics that govern the shape and length of the branches can be set to comply with kinodynamic constraints. Tree methods however can take a disproportionate time to compute paths through narrow corridors.
References for Appendix A:

26 Goodwin, S.D., Menon, S., Price, R.G., “Pathfinding in Open Terrain,” Centre for Interactive Digital Entertainment Research, School of Computer Science, University of Windsor, Windsor, Ontario, Canada, 2006
Appendix B: Additional Figures

Figure 20 A* Matlab program by Paul Premakumar
Appendix C: R* Heuristics Flow Chart

1. Inputs: Start Node, Gnode
2. Is the selected endpoint within 200m of Gnode?
   - Yes: Gnode becomes new Tnode
   - No: Terminates R*
3. Is the selected wA* path endpoint within 5% of Gnode?
   - Yes: List of Selected Paths
   - No: Endpoint becomes new start point
4. Does the number of random Tnode generations explored over a thousand?
   - Yes: List of Tnodes tried
   - No: Sort list by success rate; choose next start node
5. Random Tnode selector
6. Has the Tnode ever been selected before?
   - No: Assign Cost, Path ID & Parent Tnode ID
   - Yes: Terminates Tnode
7. List of Tnodes tried
8. wA* Path-planner
9. Path
10. List of completed and expansion limited paths
11. Assemble Grand List
12. Outputs: Paths in a y Control inputs to follow path
13. List of Tnodes used in path
14. R* Heuristics
   - Inputs
   - Heuristics
   - Output from R*
   - Processes

Final Project Report 2013, UNSW@ADFA
Appendix D: wA* Heuristics Flow Chart

wA* Heuristics
- Input from R*
- Heuristics
- Output from wA*
- Processes

Input Node: Node

1. Is Node within boundaries and not an obstacle?
   - Yes: Is Node within 100m of Node
   - No: Is Node an obstacle?

2. Does the Node already have successors?
   - Yes: Does the Node have two predecessors?
   - No: Is Node an obstacle?

3. Transfer to CLOSED list
   - Yes: TERMINATE; AVOID = 2
   - No: TERMINATE; AVOID = 1

4. Choose a new Node
   - Yes: Assign a cost to the Node based on control input
   - No: TERMINATE; AVOID = 2

5. Output Node AVOID number

6. List of Nodes
   - Yes: Is Node within boundaries?
   - No: TERMINATE; AVOID = 2
Appendix F: Results validation of Kinodynamic Snode generator by comparison with original SIMULINK AUV dynamic model created by Osama Hassan.

### Error vs Length of Time Increment

#### Figure 21 Increase in error as time increment increases

<table>
<thead>
<tr>
<th>Run time (seconds)</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change in:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x position (Dyn_select.m) (m)</td>
<td>1.3826</td>
<td>3.9998</td>
<td>7.5176</td>
<td>10.4139</td>
</tr>
<tr>
<td>x position (SIMULINK) (m)</td>
<td>1.3840</td>
<td>4.0320</td>
<td>7.7685</td>
<td>11.2722</td>
</tr>
<tr>
<td>x position (error) (m)</td>
<td>0.0014</td>
<td>0.0322</td>
<td>0.2509</td>
<td>0.8583</td>
</tr>
<tr>
<td>y position (Dyn_select.m) (m)</td>
<td>0.005084</td>
<td>0.01236</td>
<td>0.842176</td>
<td>2.408767</td>
</tr>
<tr>
<td>y position (SIMULINK) (m)</td>
<td>0.006130</td>
<td>0.01635</td>
<td>0.10790</td>
<td>0.52180</td>
</tr>
<tr>
<td>y position (Error) (m)</td>
<td>0.001046</td>
<td>0.00399</td>
<td>0.734276</td>
<td>1.886967</td>
</tr>
<tr>
<td>Orientation (Dyn_select.m) (degrees)</td>
<td>0.2107</td>
<td>1.7700</td>
<td>6.3921</td>
<td>12.3320</td>
</tr>
<tr>
<td>Orientation (SIMULINK) (degrees)</td>
<td>0.2109</td>
<td>1.7512</td>
<td>6.1850</td>
<td>13.0237</td>
</tr>
<tr>
<td>Orientation (error) (degrees)</td>
<td>0.0002</td>
<td>0.0188</td>
<td>0.2071</td>
<td>0.6917</td>
</tr>
<tr>
<td>Forward velocity (Dyn_select.m) (m/s)</td>
<td>1.3583</td>
<td>1.2608</td>
<td>1.1145</td>
<td>0.9683</td>
</tr>
<tr>
<td>Forward velocity (SIMULINK) (m/s)</td>
<td>1.3620</td>
<td>1.2889</td>
<td>1.2070</td>
<td>1.1468</td>
</tr>
<tr>
<td>Forward velocity (Error) (m/s)</td>
<td>0.003745</td>
<td>0.028134</td>
<td>0.092500</td>
<td>0.178502</td>
</tr>
<tr>
<td>Rotational velocity (Dyn_select.m) (degrees/s)</td>
<td>0.4214</td>
<td>1.17999</td>
<td>2.13069</td>
<td>2.89415</td>
</tr>
<tr>
<td>Rotational velocity (SIMULINK) (degrees/s)</td>
<td>0.4136</td>
<td>1.09512</td>
<td>1.80750</td>
<td>2.25560</td>
</tr>
<tr>
<td>Rotational velocity (Error) (degrees/s)</td>
<td>0.007798</td>
<td>0.084873</td>
<td>0.323188</td>
<td>0.638547</td>
</tr>
</tbody>
</table>

Table 1 Vehicle Response to a thrust setting of 4 RPS and a 10 degree rudder deflection given and initial starting velocity of 1.407 m/s.

dyn_select.m is the Kinodynamic Snode Generator function
Figure 22 Change in x position for 3 second time increment

Figure 23 Change in y position for 3 second time increment

Figure 24 Change in orientation for 3 second time increment

Figure 25 Change in in position after 3 second time increment (x axis corresponds to x position, y axis corresponds to y position after rotation to match SIMULINK results)

Figure 26 Kinodynamic Snode generator values for 3 second interval (Different axis coordinates to the SIMULINK results) x value corresponds to column 2, y value corresponds to column 1 and is reversed, orientation is based on an initial 90 degree starting orientation.
Table 2 SIULINK and Kinodynamic Snode Generator (dyn_select.m) results for the same (Thrust = 4 RPS and rudder deflection = 10 degrees) control inputs and an initial velocity of 1.407 m/s.
Appendix G: Paths computed by the wA* with kinodynamic Snode Generator path planner indicating versatility of path planner.

- **Figure 29**: Bug type algorithms would most likely choose the route to the right as it offers a faster route to initial obstacle clearance, but that route will trap a bug type path planner.

- **Figure 30**: The wA* path planner avoids the route to the right despite easier initial route because it can anticipate the longer total course to the right and possible dead end.

- **Figure 31**: Vertex Path Planners would increase the distance by running along the obstacles side. Voronoi Path planners would increase the distance by running along the midpoint of obstacles.

- **Figure 32**: The wA* path planner can run close or far from the obstacle enabling an optimal solution.

- **Figure 33**: Rapidly expanding Random Trees would thoroughly explore the area beneath the obstacle before choosing the relatively straight forward path through the channel.

- **Figure 34**: The wA* path planner does not waste unnecessary explorations and chooses the straight channel in front of it. Additionally the Kinodynamic Snode generator restricts the orientation of the AUV within the channel as the channel is the width of the AUV.
A potential field path planner would become trapped in the local minima of the concave obstacles as no immediate move once in the centre results in a decrease of potential.

The wA* path planner explores the concave obstacle until it finds a route out. Additionally due to the initially starting velocity of zero the path planner must use the building forward and rotational velocity to complete a right angle turn thus the requirement for a loop.

Table 3 Yellow shading indicates explored nodes and red shading indicates the position of the vehicle in six second time increments along the final path.

The lower resolution R* planner without kinodynamics. The orange points indicate the target nodes of short range paths not used in the grand path. The difficulty of following such a path is evidenced by the sharp turns including one which is a 135 degree
Figure 38 R* path planning with Kinodynamic Snode Generation
Figure 39: The wA* generated paths to random targets generated then selected from by the R* program to assemble into the grand path.
Appendix H: MACRO_search_dyn.m the R* section of the full R* path planning program

close all
clear all
cic
tic;
global CLOSED scan1 max_x max_y OPEN Obstacle_count Coef obsmap Expansion_Count

load('Coef')
Start_node = [600,950,90,0,0,0,0,1.407,0,8,1];

Gnode = [600,100];
Tnode = zeros(1,12);
W = 2;
Selected_Tnode = Start_node;

CLOSED=zeros(1,12);

scan1 = imread('1000by1000.png');
obsmap = scan1;

[max_x,max_y] = size(scan1);

OPEN=[];
Terminate = 0;
bp = 0;

MACRO_selection = [];
MACRO_selected = [];
MACRO_path = [];

GrandPATH = [];
circle_count = 0;
radius = 400;
Exp_limit = radius*3;

[Tnode_selection,Random_Tnode,Random_Tnode_size] = circle(radius);

while Terminate == 0

    if Selected_Tnode(1) == Gnode(1) && Selected_Tnode(2) == Gnode(2)
        Terminate = 1;
        break
    end

    if norm(Selected_Tnode(1:2)-Gnode)<200 %Is Gnode in range of short micro search?

        [Tnode,PATH] = MICRO_dyn(Selected_Tnode,Gnode,Exp_limit); %if in range micro search to Gnode
        q = size(MACRO_path,1) + 1;
        r = q + size(PATH,1) - 1;
        v = size(MACRO_selected,1);

    end

end
MACRO_selection =
[Tnode(1:3),v,Tnode(5:12),q,r,Tnode(13);MACRO_selection];
MACRO_path = [MACRO_path;PATH];
Selected_Tnode = Tnode(1:12); %reorder?
MACRO_selected = [MACRO_selected;MACRO_selection(1,:)];
MACRO_selection = removenode(MACRO_selection);
if Tnode(6) < 10; % close enough is good enough h is the same
as straight line distance
   Terminate =2;
   break
else
   Gnode(1:2) = Selected_Tnode(1:2); %retry from end point of
   path above
   Selected_Tnode(8) = 0; %forward velocity must be zero
   break
end
else
   random = 1;
   while random <= Random_Tnode_size(1)
      i = randi(Random_Tnode_size(2));
      Random_Tnode(random,1) = Selected_Tnode(1) +
      Tnode_selection(i,1);
      Random_Tnode(random,2) = Selected_Tnode(2) +
      Tnode_selection(2,i); %
      if Random_Tnode(random,1) > 0
         if Random_Tnode(random,1)<= max_x
            if Random_Tnode(random,2) > 0
               if Random_Tnode(random,2) <= max_y
                  if size(MACRO_selected,1) == 0
                     random = random + 1;
                  else
                     [isinlist,~] =
                     inlist(Random_Tnode(random,1:2),MACRO_selected(:,1:2));
                     if ~isinlist
                        random = random + 1;
                     end
                  end
               else
                  random = random + 1;
               end
            else
               random = random + 1;
            end
         else
            random = random + 1;
         end
      else
         random = random + 1;
      end
   end
end
product_circle = 1;
sum_circle = 0;
for random = 1:size(Random_Tnode,1) %Can be replaced by parrallel
cores
   Random_Tnode(random,(1:2)) %diagnosis
   [Tnode,PATH] =
   MICRO_dyn(Selected_Tnode,Random_Tnode(random,(1:2)),Exp_limit);
   Tnode(5) = Selected_Tnode(5) + norm(Tnode(1:2) -
   Selected_Tnode(1:2));
   Tnode(6) = norm(Tnode(1:2)-Gnode(1:2));
   Tnode(7) = Tnode(5) + W*Tnode(6); %MACRO priority
   q = size(MACRO_path,1) + 1;
   r = q + size(PATH,1) - 1;
   v = size(MACRO_selected,1);
   MACRO_selection =
   [MACRO_selection;Tnode(1:3),v,Tnode(5:12),q,r,Tnode(13)];
   MACRO_path = [MACRO_path;PATH];
product_circle = product_circle*Tnode(13);
sum_circle = sum_circle + Tnode(13);
end

bp = bp + 1 %diagnosis
circle_count = circle_count + 1;
MACRO_selection = sortrows(MACRO_selection,[15,7]); %Sort by

AVOID first then By Priority
Selected_Tnode = MACRO_selection(1,1:12);
MACRO_selected = [MACRO_selected;MACRO_selection(1,:)];
MACRO_selection = removenode(MACRO_selection); %needed to loop?
if circle_count >= 1;
circle_count = 0;
if product_circle == 0;
if radius < 400;
if sum_circle < Random_Tnode_size(1)/2
radius = radius + 20
[Tnode_selection,Random_Tnode,Random_Tnode_size] = circle(radius);
Exp_limit = radius*5;
end
else
if radius > 100
radius = radius - 20
[Tnode_selection,Random_Tnode,Random_Tnode_size] = circle(radius);
Exp_limit = radius*5;
end
end
end
if MACRO_selected(end,13) == 2
Terminate = 3;
break
end
if bp > 2000
Terminate = 3; %MACRO expansion limit
end
end

%backtracking function
flag = 4;
p_loc = size(MACRO_selected,1); %Parent node ID
dump_MACRO = [];
while flag == 4;
GrandPATH =
[MACRO_path(MACRO_selected(p_loc,13):MACRO_selected(p_loc,14),:);GrandPATH] ;
dump_MACRO = [MACRO_selected(p_loc,1:12);dump_MACRO];
if MACRO_selected(p_loc,4) ~= 0
p_loc = MACRO_selected(p_loc,4);
flag = 4;
else
flag = 3
end
end

%Braking function
while debug_MACRO(end,8) >= 0.01
    S_Snode = Dyn_select(debug_MACRO(end,1:12),1);
    dist = [ ];
    for e = 1:size(S_Snode,1)
        dist = [dist;norm([S_Snode(e,1),S_Snode(e,2)]-Gnode)];
    end
    S_Snode2 = [S_Snode,dist];
    S_Snode = sortrows(S_Snode2,[8,13]);
    debug_MACRO = [debug_MACRO;S_Snode(1,1:12)];
    GrandPATH = [GrandPATH;S_Snode(1,1:3),S_Snode(1,11:12)];
end

%Path visualisation function
scan1(Gnode(2),Gnode(1)) = 4;
scan1(Start_node(2),Start_node(1)) = 4;
c = 1;
for i = 1:size(GrandPATH,1)
    scan1(GrandPATH(i,2),GrandPATH(i,1)) = 4;
end
imagesc(scan1)

Total_time = MACRO_selected(size(MACRO_selected,1),10)/60
Run_time = toc
save('run131008.mat')
function [Tnode,PATH] = MICRO_dyn(Cnode,Tnode,Exp_limit)
%
%Parameters
w = 20;

close_enough = Exp_limit/16;

%Input
%Map black and white .png
%Cnode - selected Old Tnode
%New Tnode - randomly generated
% Cnode = [C_xpos, C_ypos, theta, Parent Node ID, g_fn, h_fn, k_fn, u , r , t , T setting, delr setting];
% Cnode = [ 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 10 , 11 , 12 ];

%scan 1 where 0 is space & 1 is obstacles
CLOSED = [];
OPEN=[];
Anode = Cnode;
imagesc(scan1)
AVOID = 0;
if Tnode(1)>0 && Tnode(1)<=max_x && Tnode(2)>0 && Tnode(2)<=max_y
next_target = 0;
if scan1(Tnode(2),Tnode(1)) ==1
next_target = 1
AVOID = 2;
else
next_target = 1; %out of bounds
AVOID = 2;
end
scan1(Cnode(2),Cnode(1)) = 2; % current = 2
Expansion_Count = 0;

while next_target == 0 %while a path exists and the path has not reached the target yet
CLOSED = [CLOSED;Cnode]; % As current node is now fully expanded, place in closed list
S_snod = Dyn_select(Cnode,6);
[B, m] = unique(S_snod(:,1:2),'rows', 'first');
Dyn_snod = [];
for q = 1:size(m,1)
    Dyn_snod = [Dyn_snod;S_snod(m(q),:)];
end
%...
s_end = size(Dyn_Snode,1);
for s = 1:s_end
    Snode = Dyn_Snode(s,:);
    if Snode(1)>0 && Snode(1)<=max_x && Snode(2)>0 && Snode(2)<=max_y %within the boundaries of the map
        Expansion_Count = Expansion_Count + 1; %AVOID count
        if scan1(Snode(2),Snodex(1)) ~= 1
            if size(CLOSED,1) >= 100
                [isinlist,~] = inlist(Snode(1:2),CLOSED(1:end-50,1:2)); %inlist limit search
            else
                [isinlist,~] = inlist(Snode(1:2),CLOSED);
            end
            if ~isinlist %check Cnode isn't an obstacle
                Snode(4) = size(CLOSED,1); %Parent Node ID - incorporate into inlist
                Snode(5) = Cnode(5) + Snode(5); % g function
                if size(OPEN,1) >= 50
                    [isinlist,index] = inlist(Snode(1:2),OPEN(1:50,1:2));
                else
                    [isinlist,index] = inlist(Snode(1:2),OPEN);
                end
                if isinlist %if a path to the same node has already be found
                    if Snode(5) < OPEN(index,5) %if this new path is better than the old path
                        flag = 1;
                        break
                    else
                        break
                    end
                else
                    Snode(6) = norm(Snode(1:2)-Tnode(1:2)); %h function
                    Snode(7) = Snode(5) + w*Snode(6); %k function
                    OPEN = [Snode;OPEN]; %put all the successors into the OPEN list
                    end
                elseif isinlist
                    end
            end
        end
    end
if ~isempty(OPEN) %as long as there is still reachable nodes
    OPEN = sortrows(OPEN,7); % sort according to lowest priority k value
    if abs(OPEN(1,1) - Tnode(1)) > close_enough
Cnode = OPEN(1,:); %the new current node is the node with the lowest priority value in the OPEN list
OPEN = removenode(OPEn); %So Cnode is not repeatedly searched
else
    if abs(OPEn(1,2) - Tnod(2)) > close_enough
        Cnode = OPEN(1,:); %the new current node is the node with the lowest priority value in the OPEN list
        OPEN = removenode(OPEn); %So Cnode is not repeatedly searched
    else
        next_target = 2 %reached the target
        Cnode = OPEN(1,:);
        CLOSED(end+1,:) = Cnode;
        break
    end
else
    next_target = 3 %no path
    AVOID = 2;
    break
end
if Expansion_Count >= Exp_limit
    next_target = 4 %Too many nodes expanded
    AVOID = 1;
    break
end
end

% Backtracking function
PATH = [];
if next_target == 2 || next_target == 4
    if norm(CLOSED(end,1:2) - Tnode(1:2)) < close_enough*1.414+10
        flag = 4;
        p_loc = size(CLOSED,1); %Parent node ID
        while flag == 4;
            PATH = [CLOSED(p_loc,1:3),CLOSED(p_loc,11:12);PATH];
            p_loc = CLOSED(p_loc,4);
            if CLOSED(p_loc,1) == Anode(1)
                if CLOSED(p_loc,2) == Anode(2)
                    flag = 3;
                    PATH = [CLOSED(p_loc,1:3),CLOSED(p_loc,11:12);PATH];
                else
                    flag = 4;
                end
            end
        end
    end
end
Tnode = [Cnode(1:3),Anode(4),Cnode(5:12),AVOID];
scan1(Tnode(2),Tnode(1)) = 3;% target = 3
else
    Tnode = [Cnode(1:3),Anode(4),Cnode(5:12),AVOID];
    scan1(Tnode(2),Tnode(1)) = 3;% target = 3
end
Appendix D: The kinodynamic Successor node generator for the full R* path planning program

```matlab
function Snode = Dyn_select(Cnode,t_increment)
    %using Cnode 3 instead of Cnode 9
    % Cnode = [C_xpos, C_ypos, theta, Parent Node ID, g_fn, h_fn, k_fn, u, r, t];

    global Coef gamma xy R du_unadj du_adj du_final

    gamma = Cnode(3)*pi/180; %clockwise rotation
    R = [cos(gamma),sin(gamma); -sin(gamma),cos(gamma)]; %anti-clockwise rotation

    snode_count = 0;
    if Cnode(8)>= 0
        Snode = zeros(88,9);
        for T = [1,2,3,4,5,6,7,8] % RPS = 1,2,3,4,5,6,-4,0
            for delr = [1,2,3,4,5,6] %0,5,10,15,20,25 degrees deflection
                snode_count = snode_count +1;
                %preallocation
                du_unadj = polyval(Coef.thrust_only(T,:),Cnode(8));
                du_adj = polyval(Coef.delr_du(delr,:),Cnode(8));
                du_final = du_unadj + du_adj;
                Snode(snode_count,8) = Cnode(8) + du_final*t_increment;
                dr = polyval(Coef.drvsu(delr,:),Cnode(8));
                %degrees/s/s Angular acceleration
                dr2 = polyval(Coef.drvsu(delr,:),Snode(snode_count,8)); %degrees/s/s Angular acceleration
                if Cnode(9)>0
                    dr3 = polyval(Coef.drvsdtheta(1,:),Cnode(9)); %degrees/s/s Angular acceleration
                    %Orientation degrees
                    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
                    else if Cnode(9) == 0
                        orient = 0.5*(dr+dr2)*t_increment;
                        orient = 0.5*(Snode(snode_count,9)+Cnode(9))*t_increment;
                        Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
                    else
                        dr3 = -polyval(Coef.drvsdtheta(1,:),Cnode(9)); %degrees/s/s Angular acceleration
                        orient = 0.5*(dr+dr2)*t_increment;
                        %Orientation degrees
                        Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
                    end
                else
                    orient = 0.5*(dr+dr2)*t_increment;
                    %Orientation degrees
                    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
                end
            end
        end
    end
```
Snode(snode_count,1) = 0.5*(Snode(snode_count,8)+Cnode(8))*cos(orient*pi/180)*t_increment; % Xpos radians/s
Snode(snode_count,2) = 0.5*(Snode(snode_count,8)+Cnode(8))*sin(-orient*pi/180)*t_increment; % Ypos radians/s sin(-veSnode(9))
Snode(snode_count,10) = Cnode(10) + t_increment;
Snode(snode_count,5) = cost_fn_dyn(T,delr,t_increment);
Snode(snode_count,11) = T;
Snode(snode_count,12) = delr;

xy = R*[Snode(snode_count,1);Snode(snode_count,2)];
Snode(snode_count,1) = round(xy(1) + Cnode(1));
Snode(snode_count,2) = round(xy(2) + Cnode(2));

end
for delr = [1,2,3,4,5,6] %0,5,10,15,20,25 degrees deflection
    %only difference dr and dr2 are negative
    snode_count = snode_count +1; %preallocation
    du_unadj = polyval(Coef.thrust_only(T,:),Cnode(8));
    du_adj = polyval(Coef.delr_du(delr,:),Cnode(8));
    du_final = du_unadj + du_adj;
    Snode(snode_count,8) = Cnode(8) + du_final*t_increment;
    dr = -polyval(Coef.drvsu(delr,:),Cnode(8)); %degrees/s/s Angular acceleration
    dr2 = -polyval(Coef.drvsu(delr,:),Snode(snode_count,8)); %degrees/s/s Angular acceleration
    if Cnode(9)>0
        dr3 = polyval(Coef.drvsdtheta(1,:),Cnode(9)); %degrees/s/s Angular acceleration
        Snnode(snode_count,9) = Cnode(9) + (0.5*(dr+dr2)+dr3)*t_increment; %degrees/s/s Angular acceleration
        orient = 0.5*(Snnode(snode_count,9)+Cnode(9))*t_increment;
    else
        if Cnode(9) == 0
            Snnode(snode_count,9) = Cnode(9) +
            0.5*(dr+dr2)*t_increment; %degres/s/s Angular acceleration
            orient = 0.5*(Snode(snode_count,9)+Cnode(9))*t_increment;
        end
        if Cnode(9) = 0
            Snnode(snode_count,9) = Cnode(9) +
            0.5*(dr+dr2)*t_increment; %degrees/s/s Angular acceleration
            orient = 0.5*(Snode(snode_count,9)+Cnode(9))*t_increment;
        else
            dr3 = -polyval(Coef.drvsdtheta(1,:),Cnode(9)); %degrees/s/s Angular acceleration
            Snnode(snode_count,9) = Cnode(9) +
            0.5*(dr+dr2)+dr3)*t_increment; %degrees/s/s Angular acceleration
            orient = 0.5*(Snode(snode_count,9)+Cnode(9))*t_increment;
        end
    end
    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
    Snode(snode_count,3) = wrapTo360(orient + Cnode(3));

Snode(snode_count,1) =
0.5*(Snode(snode_count,8)+Cnode(8))*cos(orient*pi/180)*t_increment; % Xpos %radians/s
Snode(snode_count,2) =
0.5*(Snode(snode_count,8)+Cnode(8))*sin(-orient*pi/180)*t_increment; % Ypos %radians/s sin(-veSnode(9)) anti-clockwise
Snode(snode_count,10) = Cnode(10) + t_increment;
Snode(snode_count,5) = cost_fn_dyn(T,delr,t_increment);
Snode(snode_count,11) = T;
Snode(snode_count,12) = delr;

xy = R*[Snode(snode_count,1);Snode(snode_count,2)];
Snodel(snode_count,1) = round(xy(1) + Cnode(1));
Snode(snode_count,2) = round(xy(2) + Cnode(2));
end
end
else
Snode = zeros(8,9);
for T = [1,2,3,4,5,6,7,8] % RPS = 1,2,3,4,5,6,-4,0
delr = 1; %Simplified problem

Snode_count = snode_count +1; %preallocation
du_unadj = polyval(Coef.thrust_only(T,:),Cnode(8));
du_adj = polyval(Coef.delr_du(delr,:),Cnode(8));
du_final = du_unadj + du_adj;
Snode(snode_count,8) = Cnode(8) + du_final*t_increment;

dr = polyval(Coef.drvsu(delr,:),Cnode(8)); %degrees/s/s Angular acceleration
dr2 = polyval(Coef.drvsu(delr,:),Snode(snode_count,8));
%degrees/s/s Angular acceleration
if Cnode(9)>0
    dr3 = polyval(Coef.drvsdtheta(1,:),Cnode(9)); %degrees/s/s Angular acceleration
    Snode(snode_count,9) = Cnode(9) +
    (0.5*(dr+dr2)+dr3)*t_increment; %degrees/s/s Angular acceleration
end if Cnode(9) == 0
    Snode(snode_count,9) = Cnode(9) +
    0.5*(dr+dr2)*t_increment; %degrees/s/s Angular acceleration
else
    Snode(snode_count,9) = Cnode(9) +
    0.5*(dr+dr2)*t_increment; %degrees/s/s Angular acceleration
end

orient = 0.5*(Snode(snode_count,9)+Cnode(9))*t_increment;%Orientation degrees
Snode(snode_count,3) = wrapTo360(orient + Cnode(3));
\[ \text{dr3} = -\text{polyval}(	ext{Coef.drvsdtheta}(1,:), \text{Cnode}(9)); \quad \text{% degrees/s/s} \]

**Angular acceleration**

\[
\text{Snnode(snode_count, 9)} = \text{Cnode}(9) + (0.5*(\text{dr+dr2}) + \text{dr3})*\text{t_increment}; \\
\text{orient} = 0.5*(\text{Snnode(snode_count, 9)} + \text{Cnode}(9))*\text{t_increment}; \quad \text{% Orientation degrees} \\
\text{Snnode(snode_count, 3)} = \text{wrapTo360}(\text{orient} + \text{Cnode}(3));
\]

**end**

\[
\text{Snnode(snode_count, 1)} = 0.5*(\text{Snnode(snode_count, 8)} + \text{Cnode}(8))*\cos(\text{orient} \cdot \pi / 180) \cdot \text{t_increment}; \quad \% \text{Xpos radians/s} \\
\text{Snnode(snode_count, 2)} = 0.5*(\text{Snnode(snode_count, 8)} + \text{Cnode}(8))*\sin(-\text{orient} \cdot \pi / 180) \cdot \text{t_increment}; \quad \% \text{Ypos radians/s} \\
\text{Snnode(snode_count, 10)} = \text{Cnode}(10) + \text{t_increment}; \\
\text{Snnode(snode_count, 5)} = \text{cost_fn_dyn}(\text{T}, \text{delr}, \text{t_increment}); \\
\text{Snnode(snode_count, 11)} = \text{T}; \\
\text{Snnode(snode_count, 12)} = \text{delr};
\]

\[
\text{xy} = \text{R} \cdot [\text{Snnode(snode_count, 1)}; \text{Snnode(snode_count, 2)}];
\]

\[
\text{Snnode(snode_count, 1)} = \text{round}(\text{xy}(1) + \text{Cnode}(1)); \\
\text{Snnode(snode_count, 2)} = \text{round}(\text{xy}(2) + \text{Cnode}(2));
\]

**end**

**end**
Appendix E: The supporting functions for the R* path planning program

The random target pool generator circle.m

```matlab
function [Tnode_selection,Random_Tnode,Random_Tnode_size] = circle(r)

Random_Tnode = zeros(4,2);
Random_Tnode_size = [4,20];

phi = pi/10:pi/10:2*pi;

for i = 1:20
    Tnode_selection(1,i) = round(r*cos(phi(i)));
    Tnode_selection(2,i) = round(r*sin(phi(i)));
end

if r > 200;
    r2 = r/2;
    for i = 1:10
        Tnode_selection(1,i+20) = round(r2*cos(2*phi(i)));
        Tnode_selection(2,i+20) = round(r2*sin(2*phi(i)));
        Random_Tnode = zeros(8,2);
        Random_Tnode_size = [8,30];
    end
end

% plot(Tnode_selection(1,:),Tnode_selection(2,:),'+b',0,0,'+r')
% axis([-r-r/4,r+r/4,-r-r/4,r+r/4])
end
```

The cost function for the kinodynamic Snode generator

```matlab
function g = cost_fn_dyn(T,delr,t_increment)

%Forward thruster cost
%Rudder deflection cost

% T = [1,2,3,4,5,6,7,8] % RPS = 1,2,3,4,5,6,-4,0
% delr = [1,2,3,4,5,6] % delr = 0,5,10,15,20,25 degrees deflection

sensitivity = 0.01;

if T == 1
    gt = 10;
else if T == 2
    gt = 20;
else if T == 3
    gt = 30;
else if T == 4
    gt = 40;
else if T == 5
    gt = 50;
else if T == 6
    gt = 60;
else if T == 7
    gt = 60;
else
    gt = 0;
end
end
```
end
end
end

if delr == 1
gdelr = 0;
else if delr == 2
gdelr = 0.01;
else if delr == 3
gdelr = 0.02;
else if delr == 4
gdelr = 0.03;
else if delr == 5
gdelr = 0.04;
else
gdelr = 0.06;
end
end
end
end

The coordinate searching function inlist.m used to search lists for certain x,y coordinates

function [logical,index] = inlist(node,list)
    logical = 0;
    if isempty(list)
        logical = 0;
        index = 1;
    else
        for i = 1:size(list,1)
            if node == list(i,1:2) %x=x y=y
                logical = 1.;
                index = i;
                break
            else
                logical = 0;
                index = 1;
            end
        end
    end
end

Node removal function removenode.m

function newlist = removenode(oldlist)
n = size(oldlist, 1);
newlist = oldlist(2:n,:);
Appendix F: Kinodynamic Coefficient generator vel_incr.m

format long

u = 0:0.005:1.407;

delr_only = [];

% forward acceleration as a function of forward velocity due to rudder % deflection

% delr = 0
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'K3:L2002');
delr_only(:,1:2) = sortrows(delr_only_unsorted,1);
delr0_Coef = polyfit(delr_only(:,1),delr_only(:,2),5);
delr0_no_adj = polyval(delr0_Coef,u);
delr0_Coef = delr0_Coef - delr0_Coef;

% delr = 5
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'I3:J2002');
delr5_Coef = polyfit(delr_only(:,3),delr_only(:,4),5);
delr5_no_adj = polyval(delr5_Coef,u);
delr5_adj = delr5_no_adj - delr0_no_adj;
delr5_Coef = polyfit(u, delr5_adj,5);

% delr = 10
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'G3:H2002');
delr10_Coef = polyfit(delr_only(:,5),delr_only(:,6),5);
delr10_no_adj = polyval(delr10_Coef,u);
delr10_adj = delr10_no_adj - delr0_no_adj;
delr10_Coef = polyfit(u, delr10_adj,5);

% delr = 15
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'E3:F2002');
delr15_Coef = polyfit(delr_only(:,7),delr_only(:,8),5);
delr15_no_adj = polyval(delr15_Coef,u);
delr15_adj = delr15_no_adj - delr0_no_adj;
delr15_Coef = polyfit(u, delr15_adj,5);

% delr = 20
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'C3:D2002');
delr20_Coef = polyfit(delr_only(:,9),delr_only(:,10),5);
delr20_no_adj = polyval(delr20_Coef,u);
delr20_adj = delr20_no_adj - delr0_no_adj;
delr20_Coef = polyfit(u, delr20_adj,5);

% delr = 25
delr_only_unsorted = xlsread('duvsu_delr_only.xlsx', 'Sheet1', 'A3:B2002');
% import data
importdata

delr_only(:,11:12) = sortrows(delr_only_unsorted,1);
% Sort ascending velocities

delr25_Coef = polyfit(delr_only(:,11),delr_only(:,12),5);
% Fit polynomial of degree 5 to data
delr25_no_adj = polyval(delr25_Coef,u);
delr25_adj = delr25_no_adj - delr0_no_adj
delr25_Coef = polyfit(u, delr25_adj,5);

%Isolate du due to delr

%plot du as a function of u before delr isolation
figure(1)
subplot(2,2,1),
plot(u,delr0_no_adj,'y',u,delr5_no_adj,'-r',u,delr10_no_adj,'-g',u,delr15_no_adj,'-b',u,delr20_no_adj,'-c',u,delr25_no_adj,'-m')
axis tight
xlabel('u (m/s)'
ylabel('du/dt (m/s/s)'
legend ('delr=0','delr=5','delr=10','delr=15','delr=20','delr=25')
title('Acceleration as a function of forward velocity')

delr0 = polyval(delr0_Coef,u);
delr5 = polyval(delr5_Coef,u);
delr10 = polyval(delr10_Coef,u);
delr15 = polyval(delr15_Coef,u);
delr20 = polyval(delr20_Coef,u);
delr25 = polyval(delr25_Coef,u);

%Generate plotable data points from values of u in increments of 0.005 m/s
figure(2)
subplot(2,2,2),
plot(u,delr0,'-r',u,delr5,'-g',u,delr10,'-b',u,delr15,'-y',u,delr20,'-c',u,delr25,'-m')
axis tight
xlabel('u (m/s)'
ylabel('du/dt (m/s/s)'
legend ('delr=0','delr=5','delr=10','delr=15','delr=20','delr=25')
title('Acceleration as a function of forward velocity due to delr only')

%% dr/dt as a function of forward velocity due to rudder only
u = 0:0.005:1.407;
drvsu(:,1:2) = xlsread('drvsu3.xlsx', 'drvsu', 'A3:B18');
drvsu(:,3:4) = xlsread('drvsu3.xlsx', 'drvsu', 'C3:D18');
drvsu(:,5:6) = xlsread('drvsu3.xlsx', 'drvsu', 'E3:F18');
drvsu(:,7:8) = xlsread('drvsu3.xlsx', 'drvsu', 'G3:H18');
drvsu(:,9:10) = xlsread('drvsu3.xlsx', 'drvsu', 'I3:J18');
drvsu(:,11:12) = xlsread('drvsu3.xlsx', 'drvsu', 'K3:L18');

drvsu_poly = [];
drvsu_poly(1,:) = polyfit(drvsu(:,11),drvsu(:,12),4);
drvsu_poly(2,:) = polyfit(drvsu(:,1),drvsu(:,2),4);
drvsu_poly(3,:) = polyfit(drvsu(:,3),drvsu(:,4),4);
drvsu_poly(4,:) = polyfit(drvsu(:,5),drvsu(:,6),4);
drvsu_poly(5,:) = polyfit(drvsu(:,7),drvsu(:,8),4);
drvsu_poly(6,:) = polyfit(drvsu(:,9),drvsu(:,10),4);

u = 0:1.407/100:1.407;
drvsu_val(:,1) = polyval(drvsu_poly(1,:),u);
drvsu_val(:,2) = polyval(drvsu_poly(2,:),u);
drvsu_val(:,3) = polyval(drvsu_poly(3,:),u);
drvsu_val(:,4) = polyval(drvsu_poly(4,:),u);
drvsu_val(:,5) = polyval(drvsu_poly(5,:),u);
drvsu_val(:,6) = polyval(drvsu_poly(6,:),u);

u = u';

figure(3)
plot(u,drvsu_val(:,1),'-g', u,drvsu_val(:,2),'-b', u,drvsu_val(:,3),'-y', u,drvsu_val(:,4),'-c', u,drvsu_val(:,5),'-m', u,drvsu_val(:,6),'-r')
axis tight
xlabel('u (m/s)')
ylabel('dr/dt (degrees/s/s)')
legend ('delr=0', 'delr=5', 'delr=10', 'delr=15', 'delr=20', 'delr=25')
title('Angular acceleration as a function of u')

%% dr/dt as a funtion of dtheta/dt for zero velocity and zero deflection

% RPS = 1
thrust_only(1:12000,3:4) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'A4:B12003');
% RPS = 2
thrust_only(1:12000,5:6) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'C4:D12003');
% RPS = 3
thrust_only(1:12000,7:8) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'E4:F12003');
% RPS = 4
thrust_only(1:12000,9:10) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'G4:H12003');
% RPS = 5
thrust_only(1:12000,11:12) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'I4:J12003');
% RPS = 6
thrust_only(1:12000,13:14) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'K4:L12003');
% RPS = -4
thrust_only(1:18000,15:16) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'O4:P18003');

%% Accel
eration as a function of forward velocity due to thrust only

% RPS = 1
thrust_only(1:12000,3:4) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'A4:B12003');
% RPS = 2
thrust_only(1:12000,5:6) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'C4:D12003');
% RPS = 3
thrust_only(1:12000,7:8) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'E4:F12003');
% RPS = 4
thrust_only(1:12000,9:10) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'G4:H12003');
% RPS = 5
thrust_only(1:12000,11:12) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'I4:J12003');
% RPS = 6
thrust_only(1:12000,13:14) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'K4:L12003');
% RPS = -4
thrust_only(1:18000,15:16) = xlsread('duvsu_thrust_only.xlsx', 'Sheet1', 'O4:P18003');

44
Final Project Report 2012, UNSW@ADFA
thrust_only_poly(1,:) = polyfit(thrust_only(:,1),thrust_only(:,2),5);
thrust_only_poly(2,:) = polyfit(thrust_only(:,1:12000,3),thrust_only(:,1:12000,4),5);
thrust_only_poly(3,:) = polyfit(thrust_only(:,1:12000,5),thrust_only(:,1:12000,6),5);
thrust_only_poly(4,:) = polyfit(thrust_only(:,1:12000,7),thrust_only(:,1:12000,8),5);
thrust_only_poly(5,:) = polyfit(thrust_only(:,1:12000,9),thrust_only(:,1:12000,10),5);
thrust_only_poly(6,:) = polyfit(thrust_only(:,1:12000,11),thrust_only(:,1:12000,12),5);
thrust_only_poly(7,:) = polyfit(thrust_only(:,1:12000,13),thrust_only(:,1:12000,14),5);
thrust_only_poly(8,:) = polyfit(thrust_only(:,15),thrust_only(:,16),5);

thrust_only_val(:,1) = polyval(thrust_only_poly(1,:),u);
thrust_only_val(:,2) = polyval(thrust_only_poly(2,:),u);
thrust_only_val(:,3) = polyval(thrust_only_poly(3,:),u);
thrust_only_val(:,4) = polyval(thrust_only_poly(4,:),u);
thrust_only_val(:,5) = polyval(thrust_only_poly(5,:),u);
thrust_only_val(:,6) = polyval(thrust_only_poly(6,:),u);
thrust_only_val(:,7) = polyval(thrust_only_poly(7,:),u);
thrust_only_val(:,8) = polyval(thrust_only_poly(8,:),u);

figure(5)
plot(u,thrust_only_val(:,1),'-g',u,thrust_only_val(:,2),'-b',u,thrust_only_val(:,3),'-y',u,thrust_only_val(:,4),'-c',u,thrust_only_val(:,5),'-m',u,thrust_only_val(:,6),'-r',u,thrust_only_val(:,7),'-k',u,thrust_only_val(:,8),'-k');
axis tight
xlabel('u (m/s)')
ylabel('du/dt (m/s/s)')
legend ('RPS=1','RPS=2','RPS=3','RPS=4','RPS=5','RPS=6','RPS=-4','RPS=0')
title('du/dt as a function of u due to thrust only')

%% Data export

Coef.delr_du = [[0,0,0,0,0];delr5_Coef;delr10_Coef;delr15_Coef;delr20_Coef;delr25_Coef];
Coef.drvsu = drvsu_poly;
Coef.drvsdtheta = drvsdtheta_poly;
Coef.thrust_only = thrust_only_poly;

save('
Coef.mat','Coef');
Appendix G: Miscellaneous codes

Vehicle centered path animator

close all
clear all;
clc;

%% Import Image
im=imread('scan1.png');

fhandle=figure;
imagesc(im)

%% Import Path Data
num = xlsread('Value.xls');

Time = num(:,1);
count=length(Time);

%% Initial vehicle & axis Position

%% Plot
hold on;

xpos(1) = 37;
ypos = 41;

for iCount = 2:count
    % Plot update
    xpos(iCount)=num(iCount,2);
ypos(iCount)=num(iCount,3);
P=plot(xpos,ypos,1,1);
%Axis update
% Visability arc

    xdif(iCount,1) = xpos(iCount)-xpos(iCount-1);
ydif(iCount,1) = ypos(iCount)-ypos(iCount-1);

    if xdif(iCount,1)>0
        if ydif(iCount,1)>0
            theta(iCount)=atan(ydif(iCount)/xdif(iCount)); % theta +ve
        else
            theta(iCount)=atan(ydif(iCount)/xdif(iCount)); % +/- correct
        end
    else
        if ydif(iCount,1)>0
            theta(iCount)=pi+atan(ydif(iCount)/xdif(iCount)); %+--
        else
            theta(iCount)=pi+atan(ydif(iCount)/xdif(iCount)); % -- done
        end
    end
P=plot_arc(theta(iCount)+(pi/6),theta(iCount)-(pi/6),xpos(iCount),ypos(iCount),3);

set(P,'edgecolor','b','linewidth',4)

ymin = ypos(iCount)-20;
ymax = ypos(iCount)+20;
xmin = xpos(iCount)-20;
xmax = xpos(iCount)+20;
axis([xmin,xmax,ymin,ymax]);

%Save frames for movie
M(iCount-1) = getframe;

% refreshdata
% drawnow
pause(.5);
end

plot(xpos,ypos,1,1);
axis([0,50,0,50]);
plot_arc(theta+(pi/6),theta-(pi/6),xpos,ypos,3);

%movie(M)