A Two-Dimensional Path Planning Algorithm

Richard Fox, Antonio Garcia Jr. and Michael Nelson
Department of Computer Science
The University of Texas Pan American
Edinburg, TX 78539, USA
Phone: (956) 381-3635
Email: fox@cs.panam.edu
tonyg@hiline.net
mlnelson@panam.edu

Abstract

Path planning is a necessary component of any autonomous vehicle. The task is one of finding a course that takes the vehicle from its current location to a destination avoiding obstacles that might damage or divert the vehicle. There are many forms of path planning algorithms such as those that pregenerate a path versus those that perform reactive run-time planning. This paper presents a two-dimensional pregenerating path planning algorithm that models the path as a line and compares a list of obstacles to this line to determine any collisions. Collisions are processed by selecting a point outside of the obstacle and recursively generating two new paths that avoid the collision by going around the obstacle. This paper will first describe the STESCA control architecture for autonomous vehicles and then consider path planning. A few other path planning algorithms are described. This is followed by details of the algorithm developed in this research. The paper then presents a number of examples.

Introduction

Path planning is a fundamental task of autonomous vehicles. A wide variety of algorithms exist that address different factors of path planning (see for instance [1]). These factors include whether the path should be pregenerated or generated at run time, whether the path is in a two- or three-dimensional space, the form and importance of obstacles in the space, whether the space is static or dynamic, whether other factors such as characteristics and slope of the terrain come into play, and efficiency of search.

This paper presents an algorithm for performing pregenerated path planning for an autonomous vehicle in a static, two-dimensional space, with obstacles. This research is one component of a larger project investigating the feasibility of the STESCA control strategy. STESCA (Strategic-Tactical-Execution Software Control Architecture) is an approach to providing a general-purpose control architecture for any form of autonomous vehicle.

The particular algorithm described herein is based mostly on geometric equations dealing with lines and intersections with objects. The algorithm consists of three parts: a line generator which

determines the path between two points, a collision checker which determines if the path intersects with any of the obstacles in the space, and a collision processor which alters the current path to go around the obstacle which is causing the collision. The algorithm can run in three different modes depending on the need of the autonomous vehicle in terms of efficient path versus fast path planning. This, in turn, can be dictated by the particular mission specification. This paper will offer a brief description of STESCA, followed by an examination of the path-planning algorithm, several examples, and conclude with an analysis of how the algorithm will be enhanced for actual usage by various robotic systems.

The STESCA Control Architecture

The STESCA (Software-Tactical-Execution Software Control Architecture) project being undertaken at the University of Texas - Pan American is to implement a new control approach for autonomous robotic vehicles in general [7]. In the first phase of the project, STESCA is being implemented for the Naval Postgraduate School (NPS - Monterey, CA) Phoenix Autonomous Underwater Vehicle (AUV) [8]. In the second phase of the project, STESCA will be implemented on a wheeled land-based robotic vehicle.

The three levels of STESCA are the Strategic, Tactical, and Execution levels. The top Strategic level is used to specify a mission for the vehicle. The bottom Execution level is used to control the individual components making up the vehicle. The middle Tactical level is used to translate the mission into various vehicle component commands, maintain the current state of the vehicle and the mission, and to store data collected during mission execution. Using an object-oriented approach, the major emphasis is on what each level should do, not how it should be accomplished.

An important part of STESCA is mission specification within the Strategic level. One of the primary goals is to create a system which can be used by virtually anyone with minimal training. While it may take a team of scientists and engineers to build and maintain a robotic vehicle, it should not take such a team to operate the vehicle on a daily basis. The current mission specification system [6] requires the user to specify the vehicle's path in detail. In most situations, however, the important aspects of a mission occur after the vehicle reaches some point. Rather than specifying a complete path, the user should simply be able to specify where they want the vehicle to go, without worrying about how to get it to that location. As such, the path planning algorithms discussed herein will be used to further simplify the mission specification process. It is anticipated that they will eventually be incorporated into the Tactical level for run-time path re-planning around uncharted obstacles.

A Brief Look at Some Path Planning Algorithms

One potential approach to pregenerated path planning is to model the environment space as a two-dimensional or three-dimensional array and perform a search through this space for a clear path. Each array element would represent a corresponding point in the space and contain a value indicating what is in that point. For instance, an obstacle might be denoted using one character, open space another character. If three-dimensional space is being modeled and the space is of both air and ground, then the open space should differentiate between air and ground.

There are a number of graph searching algorithms that can then be employed to find the shortest path between source and destination nodes in such a representation. For instance, open spaces could be called nodes in a graph with edges denoting the physical distances between each point and a lack of edges between two points representing an obstacle. One could then employ Dijkstra's Shortest

Path algorithm [4] to find the shortest path between the two points of interest in the graph. The generated path would not go through obstacles because Dijkstra's algorithm will only work with the available edges which are open spaces.

While this algorithm would be simple to implement, it would be very inefficient due to the size of the graph. A two-dimensional NxN space would require a graph with some N^2 nodes and as many as N^4 edges. Dijkstra's algorithm has a complexity of $O(numberofnodes^2)$ which would yield an overall complexity of $O(N^4)$ for the example space. If a three-dimensional space were used, this complexity would increase a great deal more. Other array-based searching algorithms might also be employed such as a solution used in finding the exit to a maze. Unfortunately, the complexity of such algorithms is intractable being $O(2^n)$ [3].

Another approach has been taken in an algorithm referred to as Distbug [5]. This algorithm does not use any kind of world model but instead has a simple principle of moving in a straight line towards the goal until a collision occurs. Once a collision occurs, the algorithm follows the outside of the obstacle, going around it until it can resume its straight path to the goal. This algorithm is very sensible for a reactive robot. However, the algorithm itself does not generate a path prior to run-time because it is sensor-based and therefore, obstacles are only determined when they are sensed. Two potential problems with this algorithm, from the point of view of an autonomous vehicle, are: (1) what if the obstacle is moving? and (2) what if the obstacle should not be approached at all (e.g. a mine)?

A third algorithm is based on a framed-quad-tree representation [2]. The framed-quad-tree is a variation of a general tree where each node of the tree represents a quadrant of space. Each node contains pointers to its four subregions plus a value as to whether this quadrant is passable or not. A passable quadrant has no obstacles. A quadrant that has obstacles is denoted by having its own region subdivided into four smaller quadrants, any of which might be passable or not. Passable quadrants are leaf nodes in this tree. This approach also utilizes an array which records the precise direction that the robot is taking through the space.

A general tree search algorithm can then be employed on the framed-quad-tree in order to find which quadrants should be taken to go from one point to another without collisions. Two problems with this approach are the unwieldy size of the data structure in spaces with many obstacles, and a lack of controlled search.

The size of the data structure is exponential based on the number of levels. Each node of the tree which is not a leaf node will have four children. A tree of M levels would contain on the order of 4^M nodes. Additionally, M becomes larger as the detail of the representation increases or as the size of the obstacles becomes smaller. In three-dimensional space, the tree would grow even larger due to the added dimension. A three-dimensional framed-octo-tree would have pointers to each reachable region from a given point in space. This would require a minimum of 8 pointers, and possibly 27 depending on how the space is organized.

The inefficiency of the search is due to a lack of a useful heuristic to control the search. A brute force search of the data structure would have a complexity of $O(4^M)$, that is, a complexity equivalent to the number of nodes in the tree. A heuristic search might reduce the complexity to as little as $O(M)$ if a direct path can be found from the starting point to the goal point. However, such a heuristic would have to accurately predict where any obstacle might be and avoid that quadrant of the tree entirely.

A Pregenerated Path Planning Algorithm Using Lines and Obstacles

The paper now introduces the path planning algorithm developed for the STESCA project. The algorithm pregenerates a path given a representation of the environment space. The environment is limited to two-dimensional space at the moment.

The algorithm uses two simple representations to denote the environment space. The first representation is the path itself, which is composed of a series of line segments, each denoted by two points in Cartesian space (i.e. x and y coordinates for each point). The two points are the source and destination nodes. The overall path is a collection of these line segments, in order from the overall source node to the overall destination node. The second representation is a list of obstacles in the environment space.

The algorithm is divided into three steps. The first step is to generate the straight line that separates the source and destination points. The second step is to search through the list of obstacles to determine if any intercept the line. If so, a collision occurs and the third step, obstacle avoidance, takes place. Obstacle avoidance is processed by generating a point outside of the obstacle and then generating two new line segments recursively, one from the source node to the new point and the other from the new point to the destination node. The two new line segments are then checked for collisions.

Each line segment is represented as elements composing the point-slope line formula $y = m*x+c$. In this formula, m is the slope of the line, c is the y-intercept and x and y are the coordinates of any point on the line. The line formula is computed first by determining the slope of the line connecting the two end points using the formula $(y_2 - y_1)/(x_2 - x_1)$. The y-intercept value is then computed by solving for the value c given one of the two end points and the slope.

Now that the line's components have been determined, collision processing can take place. Collisions between the line segment and any obstacles in the environment space are determined by selecting each obstacle in the area and finding if it intercepts the line at any point. Currently, all obstacles are represented as circles for simplicity where each obstacle is stored using two values, the circle's center and the circle's radius. Computing a collision is a simple matter of comparing the closest point of the line segment to the center of the circle and seeing if this distance is less than the radius plus a safety margin. If the distance is less, then the line segment comes too close or collides with the obstacle and the path must be rerouted around the obstacle.

Obstacle avoidance is performed by selecting a point outside of the obstacle and altering the path to go from the source node to that point and then on to the destination point. That is, the collision is avoided by going around the obstacle. A point is selected outside of the obstacle on the side closest to the path. There are numerous points possible. Figure 1 shows an example where two points are selected from. These two points lie beyond the obstacle at a point that would avoid the curve of the obstacle, to prevent a further collision with the same obstacle. In particular, two points are chosen that are 1 unit beyond the obstacle in the x and y directions. One point is above and beyond the obstacle, the other is below and before the obstacle. The point nearest to the collision is used is selected as the new point.

The path planner uses this new point to recursively create two new line segments from the original source node to the new node as one line, and from the new node to the original destination node as a second line. These two new segments of the path might collide with other obstacles and so both new lines are checked for collisions with all obstacles. In some situations, an oscillation between points might occur because two obstacles are too close in proximity with each other. If this happens, the system could be stuck in an infinite loop and so, when such a situation is detected, the

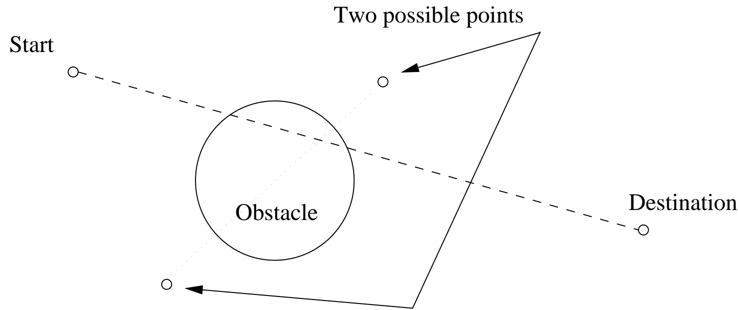

Figure 1: Selecting a New Point

system breaks the loop and returns a failure, that no path could be found.

The algorithm has three modes of operation. Each mode can generate a different path using a different amount of run-time processing depending on the particular needs of the vehicle. The algorithm as described above is mode 1. The three modes are:

1. Find the *first path*. This is the simplest mode of operation and will attempt to generate a path as quickly as possible. In this mode, when a collision occurs, a single point is generated outside of the obstacle but is still close to the original path and this point is used to generate two new line segments which will not collide with the same obstacle. This mode generates the least attempts at finding a clear path and so is very quick, $O(n^2)$ where n is the number of obstacles in the space, however, it does not find an optimal path. This mode also fails if obstacles are too close in proximity and no path can be found *between* the obstacles.
2. Find the *fewest* number of edges path. In this mode, whenever a collision occurs, the algorithm generates two new points, one above and one below the obstacle causing the collision. It then recursively generates both sets of line segments and checks each line segment for collisions. The optimal path is chosen where optimal is in terms of the fewest number of line segments.
3. Find the *shortest* distance path. This is the same mode as finding the fewest number of line segments except that the path chosen is the physically shortest in terms of distance. In most circumstances, both 2 and 3 will generate the same path.

Mode 1 is the quickest of the modes, but will often generate a path that is not as short as it could be. Mode 1 may also fail in circumstances where a path between obstacles is impossible due to lack of sufficient space between obstacles. Modes 2 and 3 can be slower depending on the number of collisions but will generate highly efficient paths. The speed of these two modes is strictly dependent on the amount of recursion due to collision processing. For every collision, the algorithm will generate twice as many paths and therefore, if there are some n collisions, there may be as many as 2^n paths generated.

A fourth mode was envisioned which would evaluate a path generated by the mode 1 version and attempt to optimize this path by removing inefficient path segments. This mode was originally planned to be implemented, but after implementing mode 2 and mode 3, it was abandoned because modes 2 and 3 will always generate better paths. However, if a situation arises where mode 1 processing should be used because processing time for a given case is limited, then mode 4 could be used. Mode 4 would attempt to take a mode 1 path and re-plot it to remove any inefficient path segments. The optimizing option will cause the path planner to consider all pairs of path segments

and attempt to simplify the overall path by removing one or more of the segments. While this mode would slower than mode 1, it would often be faster than mode 2 and 3 and deliver a better path than mode 1.

Examples

In this section, four example spaces are presented to demonstrate the path planner's output running in the different modes. For each example, the three implemented modes are demonstrated. Each figure shows the source and destination points, the obstacles, and the paths generated by the different modes. The dashed line in each figure represents the original path as if there were no obstacles. In examples 1 and 2, mode 2 and mode 3 generate the same path. In these examples (figures 2 and 3 respectively), the dotted line segments represent the mode 1 path and the solid line segments represent the mode 2 and mode 3 paths. In example 3, mode 1 and mode 3 generate the same path (in figure 4, this is denoted using dotted line segments) and mode 2 generates a different path (denoted as solid line segments). In example 4, mode 1 and mode 2 generate the same path (in figure 5, this is denoted using dotted line segments) and mode 3 generates a different path (denoted as solid line segments).

Example 1

The first example is provided in figure 2, where the path is to go from point (1, 1) to point (15, 15) with three obstacles at (2, 5) of radius 1, (6, 5) of radius 2 and (10, 13) of radius 2. The original path would be the simple line from (1, 1) to (15, 15). However, there is a collision with the obstacle at (6, 5).

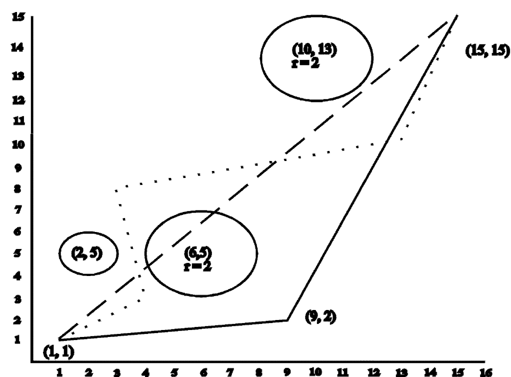

Figure 2: Example 1

In mode 1, the planner attempts to create a new course above the circle at (6, 5). The decision to go above the obstacle is made because the collision is towards the top of the obstacle and going above it would require less effort than going below it. The path planner generates two new lines, from (1, 1) to (3, 8) and from (3, 8) to (15, 15).

Next, the path segment from (1, 1) to (3, 8) is checked for collisions. A collision occurs with the obstacle at (2, 5). The planner now generates a new point, below (to the right of) the obstacle at (4, 3). There are now three path segments, from (1, 1) to (4, 3), from (4, 3) to (3, 8) and from (3, 8) to (15, 15). The first path segment, from (1, 1) to (4, 3), has no collisions. The second path segment, from (4, 3) to (3, 8) has no collisions. However, the last path segment, from (3, 8) to (15, 15) has

a collision with the obstacle at (10, 13). The planner decides to go below this obstacle creating a new point at (13, 10). The final path then goes from (1, 1) to (4, 3) to (3, 8) to (13, 10) to (15, 15). None of these path segments have collisions and therefore a path has been found.

Mode 2 and mode 3 generate an identical path which is more efficient than the path generated by mode 1. In mode 1, the first collision was avoided by selecting a point above the obstacle. This is the wrong decision as it leads to another collision. Selecting a point below this first obstacle would simplify the path. In modes 2 and 3, the collision is processed using both points and selecting whichever final path has fewer segments and is shorter respectively. This path goes from (1, 1) to (9, 2) to (15, 15).

The reason that the mode 2 and 3 path is so much simpler than that one determined using mode 1 is because mode 2 and mode 3 made an attempt to derive a path both above and below the first collision with the obstacle at (6, 5). Going above the obstacle, which common sense would dictate to be better, creates several other collisions. Going below the obstacle finds a path free of other obstacles and therefore no other collisions.

Example 2

Example 2, shown in figure 3, has a path from (1, 2) to (11, 2) with obstacles at (6, 6), (6, 2) and (9, 3) of radius 1. The original path collides with the obstacle at (6, 2). In mode 1, the collision is avoided by going above the obstacle with a new point at (6, 4). The new path to (6, 4) to (11, 2) collides with the obstacle at (9, 3). To avoid this collision, a new point is selected at (11, 2). The final path from mode 1 is from (1, 2) to (6, 4) to (11, 5) to (11, 2).

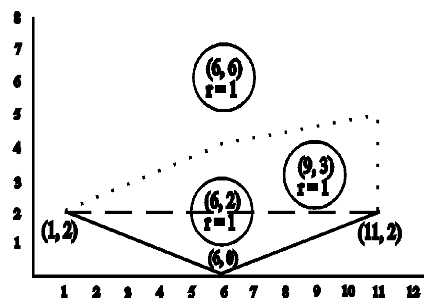

Figure 3: Example 2

In modes 2 and 3, the original collision is avoided by going below the obstacle, selecting a point at (6, 0). The two new line segments, from (1, 2) to (6, 0) and from (6, 0) to (11, 2) have no further collisions.

Example 3

The third example, given in figure 4, is a simpler space with a path from (1, 3) to (12, 11) and two obstacles, one at (6, 8) with a radius of 3 and one at (4, 3) with a radius of 1. The initial path collides with the obstacle at (6, 8).

In mode 1, the newly generated point is below this obstacle, at (10, 4). The new line segment from (1, 3) to (10, 4) now collides with the obstacle at (4, 3). A third point is chosen, at (2, 5) to avoid this obstacle. The final mode 1 path is from (1, 3) to (2, 5) to (10, 4) to (12, 11). Mode

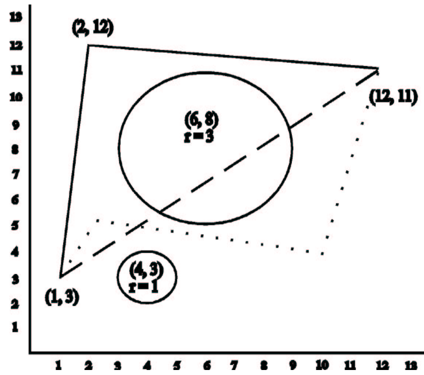

Figure 4: Example 3

2, however, generates a new point above the obstacle at (6, 8), providing a path of only two line segments from (1, 3) to (2, 12) to (12, 11). Mode 3 selects the same path as mode 1 because it has a shorter distance, approximately 17.6, then any other path (the mode 2 path has a distance of approximately 19).

Example 4

Example 4, shown in figure 5, has a path from (1, 1) to (11, 6) with obstacles at (2, 5), (5, 7), (6, 4), (8, 4) and (6, 1), all with a radius of 1.

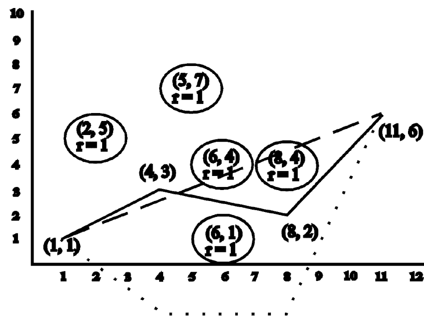

Figure 5: Example 4

The original path collides with the obstacle at (6, 4). The collision occurs towards the bottom of the obstacle and so the path is rerouted below the obstacle. The new point chosen is actually inside of the obstacle at (6, 1) and therefore, another new point is chosen, at (8, -1). The path now goes from (1, 1) to (8, -1). Another collision occurs with the obstacle at (6, 1) and so another new point is selected, this time at (4, -1). The resulting path from mode 1 goes from (1, 1) to (4, -1) to (8, -1) to (11, 6).

The mode 3 path, instead, selects a course that weaves between the obstacles from (1, 1) to (4, 3) to (8, 2) to (11, 6). The overall distance for the mode 1 path is about 16.2 whereas the overall distance for the mode 3 path is only 12. 7. The mode 2 path is the same as mode 1, although it could as equally have been the same as mode 3 since they have an equal number of path segments. Mode 2 selects the first path that has the fewest number of path segments, which was the path generated by mode 1.

Future Modifications

The current algorithm is flawed in four ways. First, it does not take into account any heuristics in its search. For instance, there may be knowledge that will help the path planner select one point over another to avoid an obstacle. If the terrain is rough on the north side of an obstacle, the path should go to the south side.

Second, the algorithm treats all obstacles the same - two-dimensional circles. Obstacles can come in any shape. The choice for using circles as representations was made to simplify the initial algorithm. Changing obstacles to other shapes will require a modification of the obstacle collision function.

Third, the algorithm only works in two-dimensional space. To be realistic, the algorithm must work in three-dimensional space. Even a land based robot would require three dimensions because the robot may be too large to fit under a structure or the landscape itself might be sloped. Without modeling the third dimension, the path planner is limited to unrealistic environment spaces.

Fourth, the algorithm only works on a static environment without the benefits of reactive planning. This particular problem, however, is not necessarily a flaw of the algorithm and can be handled using other algorithms within the STESCA model.

The first three of these flaws are being addressed in order to enhance the algorithm. To address the first flaw, a heuristic can be applied to immediately rule out a point generated to avoid an obstacle. For instance, if the terrain of that point is rough or unstable, the point can be discarded and another point can be generated. In order to address the second flaw, other obstacle shapes will be incorporated and the collision detector will be modified. Circles were chosen because of a desire for a uniform shape in the initial implementation. Other shapes such as squares, rectangles and ovals will be added. More exotic shapes will require greater care. Finally, to address the increase from two to three dimensions will require adding the ability to represent the line and all obstacles as three-dimensional objects. This modification should be straightforward.

Rather than addressing the fourth flaw of the algorithm, the static nature, by modifying the algorithm itself, a different approach is envisioned. The path planning algorithm is intended as a pregenerating path planner, and therefore, does not require reactive planning. However, the autonomous robotic vehicles will require reactive planning. And therefore, a separate component of the robot vehicle will take up this burden. Having generated a path for the robot to take, sensors will alert the tactical level of new obstacles and a reactive planning mechanism will decide how to react, whether to stop or reroute the course, or to generate a completely new path through the space given updated information.

Conclusion

This paper has presented a path planning algorithm for use by an autonomous vehicle operating in two-dimensional space. This research is one component of a larger effort to develop a general-purpose autonomous vehicle architecture known as STESCA. The path planning algorithm uses two representations, a line and a list of obstacles in the space. A line is formed from the two end points of the path and the obstacles are examined to see if any intercept the line. If found, then collision processing finds a point outside of the collision and recursively creates two new lines segments. The algorithm works in one of three modes depending on the need to find a path quickly versus an efficient path. The algorithm is not currently sufficient due to several limitations. Changes envisioned for

the algorithm are to move it to a three-dimensional space, to allow for heuristics and to have more realistically shaped obstacles.

Acknowledgments

Partial funding for this project has been provided under NASA grant OEOP Faculty Award for Research [6]. Additional funding has been obtained from a University of Texas - Pan American Faculty Research Grant.

References

[1] *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems, Volumes 1-3.* IEEE Computer Society Press, 1995.

[2] D. Z. Chen, R. J. Szczerba, and J. J. Urhan Jr. Planning conditional shortest paths through an unknown environment: A framed-quadtree approach. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and System Human Interaction and Cooperation*, volume 3, pages 33–38. IEEE Press, 1995.

[3] N. Dale and S. C. Lilly. *Pascal Plus Data Structures.* Heath Publishing, fourth edition edition, 1995.

[4] E. W. Dijkstra. A note on two problems in connection with graphs. pages 269–272.

[5] I. Kamon and E. Rivlin. Sesnory based motion planning with global proofs. In *Proceedings of the 1995 IEEE/RSJ International Conference on Intelligent Robots and System Human Interaction and Cooperation*, volume 2, pages 435–440. IEEE Press, 1995.

[6] M. L. Nelson. Object-oriented software control architecture for robotic vehicles. Technical report, The University of Texas - Pan American Reseach Proposal - NASA Proposal NRA 96-OEOP-1, Edinburg, TX, Feb., 1996.

[7] M. L. Nelson and V. Rohn. Mission specification for autonomous underwater vehicles. In *Proceedings of Oceans '96*, pages 407–410. MTS/IEEE Press, September, 1996.

[8] NPS AUV Web Site. *http : //www.cs.nps.navy.mil/research/auv/about_auv.html.* Technical report, Naval Postgraduate School, Monterey, CA, 1997.