# Path Planning for an Autonomous Vehicle

by

**Scott Douglas McKeever**

B.S. Operations Research
United States Air Force Academy, 1998

SUBMITTED TO THE SLOAN SCHOOL OF MANAGEMENT IN
PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

**MASTER OF SCIENCE**
at the
**MASSACHUSETTS INSTITUTE OF TECHNOLOGY**
June 2000

Signature of Author:______________________
Operations Research Center
12 May 2000

Certified by:______________________
Dr. Michael J. Ricard
Charles Stark Draper Laboratory
Technical Supervisor

Certified by:______________________
Professor Georgia Perakis
Assistant Professor of Operations Research
Thesis Advisor

Accepted by:______________________
Cynthia Barnhart
Associate Professor of Civil Engineering
Co-Director, Operations Research Center

# Path Planning for an Autonomous Vehicle

by

Scott Douglas McKeever

Submitted to the Sloan School of Management
on May 12, 2000, in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Operations Research

## ABSTRACT

The desire for highly capable unmanned autonomous vehicles (UAVs), has necessitated the need for more research into the problems faced by these vehicles. One classic problem faced by UAVs concerns how the vehicle should traverse its environment in order to leave the current position and arrive at a desired location. The path to this goal location must maneuver the vehicle around any obstacles and reach the goal with minimal cost. A variant of this problem tasks the UAV with tracking a moving target. In this manner the UAVs trajectory is updated through time to reflect changes in the target's location.

The specific mission addressed in this thesis, is the track and trail mission. This mission tasks a UAV with acquiring a target vehicle and tracking the vehicle for an indefinite period of time. The goal of this mission is not to intercept the vehicle, but to follow the target from a certain standoff distance. One can imagine many applications of this mission. One such application envisioned by the United States Navy deals with an unmanned underwater vehicle (UUV), tracking an enemy submarine. In addition, marine biologists could use such a capability to allow a UUV to follow and record valuable information on certain species.

Planning these paths is conceptualized as a series of network shortest path problems. This thesis focuses on planning paths in the plane where the state of the vehicle is defined only by its position in space. In addition, a trajectory smoothing or path-smoothing component is addressed to eliminate any slope discontinuities as a result of the shortest path algorithms. A framework for the moving target shortest path problem is created. The resulting path planner is capable of performing the stated mission. A detailed simulation of the path planner operating on a UUV in an underwater environment is created in order to test the planner's performance. Different variants of the path planner are created in order to deal with different problem parameters. The resulting path planner is shown to be adaptable to these different conditions and effective at tracking a moving target.

Technical Supervisor: Michael J. Ricard
Title: Senior Member of the Technical Staff, Charles Stark Draper Laboratory

Thesis Advisor: Georgia Perakis
Title: Assistant Professor of Operations Research

## ACKNOWLEDGEMENTS

I would like to thank to extend my sincerest gratitude to all of those who have helped me attain my goals and dreams throughout my life. Without their inspiration and support this thesis would not have been possible.

First and foremost, I would like to thank my advisor at Draper Laboratory, Mike Ricard, for providing me with the opportunity to continue my education at MIT. His insight and experience in the subject matter, enabled difficult areas of research to be overcome with ease. At times he has sacrificed his own priorities for the sake of my own. Mike always seemed to have the answers, and now I hope that I have learned just a few those answers. Thank you, Mike.

I also offer my gratitude to my advisor at MIT, Georgia Perakis. Her wisdom and dedication to her students, is a testament to us all. She provided many comments and suggestions that contributed to the development of this thesis.

All of my friends here in Boston have made these past two years some of the best in my life. My roommates, Nick, George, and Jim are all due much thanks for putting up with me for the past 2 years. Nick, is due even more acknowledgement. He has had to put up with me for the past 4 years. As roommates, officemates, and best friends, Nick always lends a helping hand when one is needed. To Jacki, I also say thanks. I think it was the phone calls late at night, while I was busy writing my thesis, that helped make the time pass with a little less pain. Her unending encouragement made this thesis a reality.

Finally, I would like to thank my family for providing years of guidance and support. With their help I have been able to realize so many of my dreams and goals.

This thesis was prepared at the Charles Stark Draper Laboratory, Inc. under Internal Research and Development #00-1-2068, "*UUV Path Planning in Dynamic Environments*", and contract N66604-97-0-0922 from the Naval Undersea Warfare Center Division, Newport. Publication of this thesis does not constitute approval by Draper or the sponsoring agency of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

[illegible]

Scott D McKeever, 2 Lt., USAF 12 May 2000

# Table of Contents

## Table of Figures

# Nomenclature

| | |
|---|---|
| $G$: | The notation of a network or graph. $G=(N,A)$ |
| $N$: | The set of nodes in a network. Individual nodes are represented as $i \in N$ |
| $A$: | The set of arcs in a network. The individual arcs are denoted by $(i,j) \in A$ For instance, the arc from node $i$ to node $j$ is defined as $(i,j)$. The arc from node $j$ to node $i$ is $(j,i)$ |
| $n$: | The number of nodes in a network (unless otherwise specified) |
| $m$: | The number of arcs in a network (unless otherwise specified) |
| $c_{i,j}$: | The cost of arc $(i,j)$. The cost of moving from node $i$ to node $j$ |
| $p$: | The shortest path. Collection of nodes and arcs |
| $(x,y)$: | Location in cartesian coordinates |
| $Z$: | The map of the environment |
| $Z_{res}$ | Resolution (meters). The smallest possible grid cell in a map is $Z_{res}$ x $Z_{res}$. For a fixed grid map all cells are $Z_{res}$ x $Z_{res}$ |

# Chapter 1

# Introduction

The objective of this thesis is to provide an efficient path planner for an Unmanned Autonomous Vehicle (UAV), which is tasked with tracking a moving target. This problem is characterized in such a manner that allows a succession of network shortest paths to meet the desired goals of the path planner. A simulation of the path planner as it applies to an actual vehicle is created and helps to validate the performance of the path planner.

## 1.1 Problem Motivation

The rapid increase in computer technology has carried over into the burgeoning industry of autonomous vehicles. The word autonomy is a buzzword in many of the fields of engineering. An autonomous system is one that can perform its designated task without the aid of a human or other intelligent operator. The power of computing has allowed engineered systems to reduce their dependence on the "human in the loop." The promise and implementation of autonomous systems is such that humans should live a safer, more efficient existence. Specifically, autonomous vehicles can perform tasks in environments too hostile to humans, or tasks that are too tedious. In fact, the United States Air Force has already employed the use of an unmanned airplane, the Predator, to perform extended surveillance type missions. The Predator has performed well in numerous conflicts in the past few years and has provided valuable data to the commanders in the field of operations. The type of mission performed by the Predator could be very dangerous to a pilot, and the length of flight duration could prove too long for a human pilot. Although the Predator is not fully autonomous, some of its mission is carried out in an autonomous fashion. The Predator serves as an example of a successful unmanned vehicle and motivates a future with more unmanned vehicles.

A problem frequently encountered in UAVs is the path planning problem. Given the current location of the vehicle and a desired location, a UAV typically needs to discover how to traverse its environment to the desired location in a cost optimal fashion. Typically the environment is composed of a set of threats and/or obstacles. In addition, certain constraints may be placed on the vehicle's motion. The measure of cost could be time, distance, fuel use, or vehicle safety. Therefore, it is the role of the path planner to generate a trajectory from the current point to the desired point while avoiding the threats and/or obstacles. In addition, this path should be a path of minimum cost. However, many times there exists a tradeoff in cost optimality and computation time. Due to limited computing power, and operational constraints on time allowed to plan, the path planner must be efficient in its task, even at the sacrifice of optimality.

## 1.2 General Problem Statement

The specific problem addressed by this thesis, is the track and trail mission as outlined in the executive summary of *The Navy Unmanned Undersea Vehicle (UUV) Master Plan* [5]. The operational environment is underwater, and the vehicle in this type of environment is typically referred to as a UUV. The track and trail mission tasks a UUV with acquiring a target as it leaves its port and following the target for an indefinite period of time. While the approach is designed with this application in mind, the path planner should be able to track other types of moving targets.

The path planner as developed in this thesis is separated into two main functional components. The first functional component finds a shortest path in a network, and the second component focuses on smoothing that path. The smoothing is performed so that the UAV can follow the shortest path as generated by the network shortest path. Both of these functions operate within the path planner framework, which is tasked with maintaining the correct data interchange between these units.

In order to simulate the path planner a specific test environment of operation is chosen. Through the cooperation of the Naval Underwater Warfare Center, bathymetry data of Narragansett Bay, Rhode Island, was made available. The data is also stored at the National Geophysical Data Center and is publicly available [2]. This data is then used to characterize the test environment.

## 1.3 Unmanned Autonomous Vehicles

### 1.3.1 Brief Description

In creating an autonomous vehicle, one usually attempts to model a system that relies on a human operator. The difficulty in creating true autonomy lies in the ability to transform the thinking and actions of the human operator into an effective set of behaviors and rules by which the autonomous system will abide. In the case of a submarine, human operators in the past have been vital to its operation. Some subsystems within the submarine consist of navigation, mission planning, task execution, propulsion, weapons deployment, etc... Each of these subsystems relies heavily on the "human in the loop" and must be carefully modeled in the creation of the autonomous vehicle.

A UAV can be thought of in context of a hierarchy. The highest level process can be thought of as the mission planner. The mission planner controls all the major functional units of the UAV. These functional units could include navigation, situational awareness, control, and plan generation. These functional units then control the processes that are internal to these functions. For instance, map maintenance may be a function internal to the situational awareness block. The mission of a UAV is planned, and carried out on a high level by the mission planner. As the mission progresses through time the mission planner controls how each of these functional units are invoked.

Another way to characterize how a UAV works is to think of its operation in terms of the following diagram.

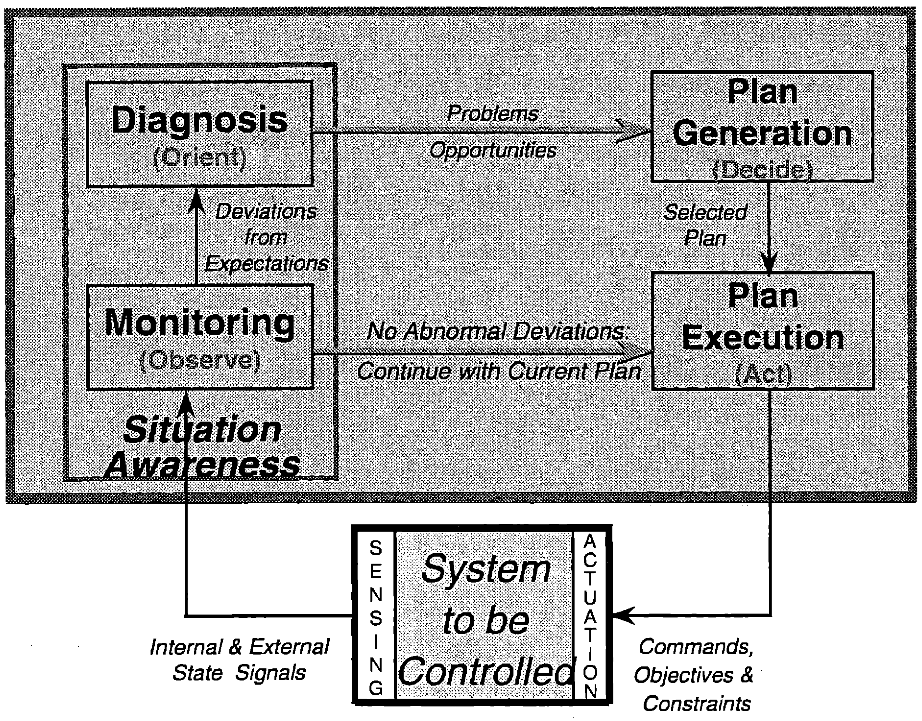


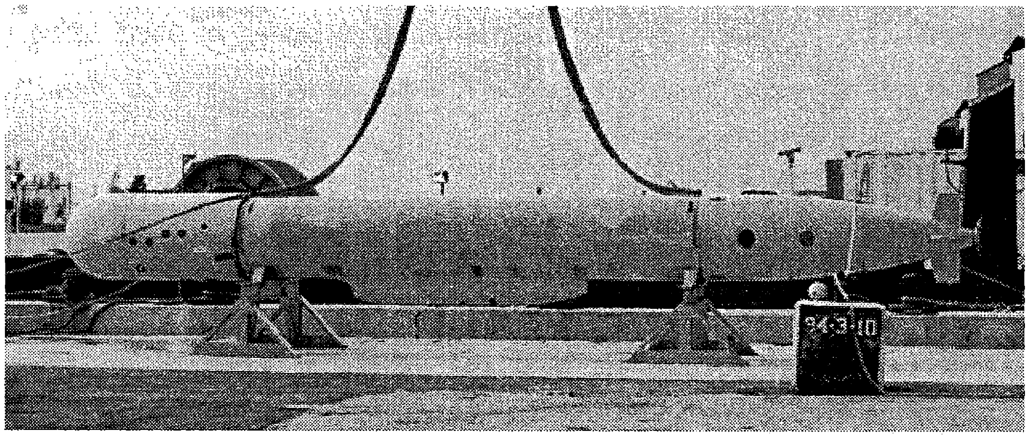
**Figure 1.1 OODA LOOP**

This diagram is known as the OODA loop (Observe, Orient, Decide, and Act), and is frequently used within the military to assess the ability of a unit to successfully carry out its mission [8]. It attempts to organize actions performed by the unit into one of four major blocks. These four blocks (Orient, Observe, Decide, and Act), are equally applicable to a UAV. If the UAV can successfully carry out this loop in the context of its mission, then the mission will be successful.

The monitoring (Observe) block can be characterized as the vehicle's ability to correctly incorporate information about its state and environment. The diagnosis (Orient) block corresponds to the vehicle deciding which actions need to be performed in the future, given the current data from the monitoring block. The plan generation (Decide) block of the OODA loop is the focus of this thesis. In this block, the actual plans of action are created. Depending on the results of the diagnosis block, certain functional units within the plan generation block may be activated. In fact, the path planner fits completely within this block. After a feasible plan is generated, the plan execution (Act) block sends the commands to the vehicle actuators. As the plan is executed, the sensors constantly update the state and/or condition of the vehicle. If the monitoring block senses the need for a new plan, then the entire loop is invoked once again.

### 1.3.2 AMMT Example

Under a program called Autonomous Minehunting and Mapping Technologies (AMMT), Charles Stark Draper Laboratory built the UUV shown it the picture below:



This vehicle demonstrated the ability to operate in an unknown environment, identify various underwater mines, and generate a map of the environment. The planner was not concerned with moving target identification, and for a given invocation of the planner the data (i.e. the map) was considered fixed. The vehicle was designed to plan in relation to position and vehicle heading. Many technical challenges remained after producing this vehicle. Namely, improvements to the trajectory generation and the production and maintenance of the map and underlying network were singled out as needed advances [10]. These lessons learned motivated the use of a 2-D grid map as both the map and the underlying network. This choice simplifies both the trajectory generation and the maintenance and production of the map and network.

## 1.4 Network Shortest Path

The moving target path-planning problem can be approached many ways. This thesis utilizes shortest path algorithms to perform a majority of the path planner's functionality. The first step taken is converting the environment into a network representation. Next, a shortest path is found on the network in relation to a cost metric of choice. Finally, the path planner performs any necessary changes to the path, and passes commands to the vehicle's actuators.

A breadth of algorithms exists for solving the network shortest path problem. This thesis gives an overview of the following algorithms: Dijkstra, A*, and D*. The algorithm implemented in the simulation is the D* algorithm with Dijkstra like performance characteristics.

The moving target scenario is accomplished as a series of successive shortest paths. The first path is computed from the UAVs initial position, and all other plans are computed from a target location to the next target location. The path from seeker to target is updated, as information from the successive shortest path computations change. The details for how the path is created are given in Chapter 5.

## 1.5 Contributions

This thesis demonstrates the applicability of solving a series of network shortest path problems in hopes of tracking a moving target. The simulation as discussed in the final chapters, demonstrates the usefulness of the techniques developed for the path planner. The success of the approach should motivate future UAVs, tasked with tracking a moving target, to employ such a path planner.

The contributions are listed below:

1) Problem Formulation
    - Creating the network as an overlay on the map
    - Transforming the moving target tracking problem into a series of shortest path algorithms
2) Path Smoothing
    - Solving the smoothing problem as a linearly constrained quadratic program
    - Convex Approximation of Curvature Formulation

    - 2-Turn Procedure

3) Implementation of Path Planner

    - Integration of Path Smoothing with Shortest Paths
    - Demonstrates the capabilities of the Path Planner

The path planner designed in this thesis may be used in the development of an on-board path planner for future Navy UUVs. The work performed in this thesis demonstrates the ability for an autonomous vehicle to track a moving target, while traversing an environment with obstacles.

## 1.6 Organization

The main body of the thesis is divided into four chapters. The first two chapters provide the background necessary to develop the moving target path planner. The first background chapter gives an overview of network shortest paths, how the environment can be represented as a digital map, and the creation of the network form this digital map. The following chapter details a few approaches to smoothing the paths as output by shortest path algorithms. The next two chapters describe the path planning approach and provide some computational results. Chapter 4, gives a detailed description of the path planning algorithm. Chapter 5 takes a detailed look into how the path planner performs on specific test cases. Chapters 4, and 5 constitute the majority of the contribution made by this thesis. Finally, a conclusion chapter provides a summary of the path planning method as well as recommendations for future work.

# Chapter 2

# Network Shortest Path

## 2.1 Overview

This chapter provides background on the network shortest path problem and algorithms. In addition, the characterization of a vehicle's environment and the transformation of that characterization into a network structure are discussed. This representation of the environment as a network allows the path planning problem to be interpreted as a shortest path problem. Furthermore, this chapter provides the motivation for an efficient moving target planner. Lastly, a section is devoted to issues of implementing a shortest path algorithm onboard an actual vehicle. This entire chapter sets the stage for the need for trajectory smoothing (Chapter 4), and an efficient moving target planner (Chapter 5).

## 2.2 General Problem Description

The shortest path problem is one of the most common and useful problems in all types of network applications. Applications of network shortest paths are seen in many areas, such as UAV path planning, internet networking, traffic networks, and large-scale supply networks to mention just a few [7]. A shortest path in a graph, *G*, is defined as the minimum cost path from some node *s* (source), to some node *t* (target). A path, $p_{[i,j]}$, is the collection of arcs and nodes from node *i* to node *j*. The total cost of the path, *p*, is the sum of all $c_{ij}$ such that $i,j \in p$ . Formally, the formulation looks as follows:

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}$$
$$\textit{subject to}:$$
$$\sum_j x_{ij} - \sum_j x_{ji} = b_i \qquad \forall i \in N$$
$$b_i = \begin{cases} 1, \textit{ if } i=s, \\ \text{-}1, \textit{ if } i=t, \\ 0, \textit{ otherwise} \end{cases} \tag{2-1}$$
$$x_{ij} \in \{0,1\} \quad \forall (i,j) \in A$$

In (2-1), $x_{ij} = 1 \textit{ if } (i,j) \in \textit{path } p, x=0 \textit{ if } (i,j) \notin \textit{path}$. The shortest path problem (or min-cost path problem) is a special case of the min-cost flow problem with the net flow at each node, *b(i)*, equal to zero except at the start and goal nodes [7].

The above formulation can be generalized to the case where the source and target move through time. The shortest path formulation,

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij}^{\bar{T}}$$
$$\textit{subject to}:$$
$$\sum_j x_{ij}^{\bar{T}} - \sum_j x_{ij}^{\bar{T}} = b_i^{\bar{T}} \qquad \forall i \in N \tag{2-2}$$
$$b_i^{\bar{T}} = \begin{cases} 1, \textit{ if } i=s_{\bar{T}}, \\ \text{-}1, \textit{ if } i=t_{\bar{T}}, \\ 0, \textit{ otherwise} \end{cases}$$
$$x_{ij}^{\bar{T}} \in \{0,1\} \quad \forall (i,j) \in A$$

specifies the shortest path in *G* at a specific time, $T = \bar{T}$. This formulation assumes *c<sub>ij</sub>* does not change through time. This model is essential the static shortest path at a snapshot in time. In this model no restrictions are placed on the source and target information.

In the case of computing incremental shortest paths to a moving target, more information is known. First we discretize time, so that $T \in \{1,..T_f,\}$ and we let $P^T$ = shortest path at time $T$. Assume that as a seeker travels along $P^1$, information about $t_2$ (target at $T_2$) is discovered. A new shortest path from the current location to the new target must then be found. The path will be computed form a new source to a new goal. It is known that $s^T \in P^{T-1}$ . In the case of tracking a moving target, the most general assumption about the target is chosen. Basically, no information about the target's position at $T+1$ , is available at time $T$ (or before). Essentially, the problem is still a static shortest path problem, at snapshots in time.

## 2.3 Map and Network Representation

All autonomous vehicles must have the ability to interface with a discrete representation of their environment. The type of representation chosen can have a large affect on the underlying network representation, and path planning performance. Efficiently computing shortest paths relies on a network representation of the variables and data as shown in (2-1). Frequently, the discrete representation of the environment, herein termed the map, and its structure can be used as a representation of the network. Using the map in this manner eliminates the need to transform data from the map into a separate storage for the network.

### 2.3.1 Map Overview

Many vehicles operate in relatively unknown environments and are responsible for sensing the environment and creating a discrete representation of that environment. However other vehicles, may operate in well-know environments and have access to excellent a-priori representations of that environment. This discrete representation is herein termed the map. In the context of path planning, the environment is usually analyzed in terms of traversal cost. An element of the environment with infinite traversal cost, is an obstacle. In addition, varying terrain may incur different traversal costs. For example, suppose the vehicle of interest is a car. The car should know the difference between a highway and a dirt road. Both types of terrain may allow travel, however the highway is much faster and cheaper to traverse. Due to mission requirements, the vehicle may need to store all types of information about its environment. For example, in AMMT, mine location was paramount to the mission. However, in the context of path planning, information that is not pertinent to generating a path will be ignored.

How is the environment discretized? The answer to this question is dependent on the operational environment, vehicle capabilities, and the mission. However, almost all applications attempt to divide the environment into polygons in the 2-D representation, or polyhedrons in the 3-D representation. Most 2-D discrete representations split the environment into a set of squares, or grid cells. In a 3-D world, the extension to a cube is made and the environment is now represented as a 3-D matrix. Each element of the matrix then stores a cost. Often each cell can be represented as either a 0 or 1. A 1 value represents an obstacle, and a 0 represents free space. In this setting all cells with non-zero value are equal in terms of traversal cost. For the remainder of this thesis, the 0-1 representation is used. In the most general setting, the value stored in each cell should be set to some positive number, as a measure of relative traversal cost. If the 0-1 representation is used, and the environment is relatively obstacle free much of the storage is wasted in storing free space. For a more efficient representation of free space see section 2.3.3 on quadtrees.

Another consideration in map storage is the resolution. Frequently, the resolution is limited by on-board memory, and the path planner's algorithmic performance. Sometimes a local map with better resolution may be needed to better represent the environment close to the vehicle. Clearly, accurately representing an obstacle 10 km from a vehicle is less important than accurately representing an obstacle 10 meters from the vehicle. The global map can be used for path planning, while the local map can be used as a means to update the global map and as information for obstacle avoidance. This approach is taken by many unmanned vehicle projects, including the Mars Autonomy Program, at Carnegie Mellon University. Obstacle avoidance behaviors are alternatives to planning methodologies. These behaviors are largely reactive in nature. Even though obstacle avoidance (or emergency plans) are an important behavior on-board a UAV, they are not developed in this thesis.

### 2.3.2 Fixed Grid

The map as a fixed grid is an intuitive choice of environment representation. Essentially a matrix represents the environment with the elements of the matrix representing the center points of each grid cell. In the context of an underwater application, each grid cell can store the depth of the ocean at that location. If one wants to know if an obstacle exists at a certain depth, each grid cell is checked for the stored depth at that location. Therefore, at a certain depth, a binary map can be stored using the following argument. A grid cell *(i,j)* is defined as the square of size $Z_{res} \times Z_{res}$ centered at location $(Z_{res} * i, Z_{res} * j)$, where $Z_{res}$ is the resolution of the map. Obstacles are stored as a 1, such that if an obstacle exists within cell *(i,j)*, $Z(i,j) = 1$. Likewise if the cell *(i,j)* is free of obstacles, then $Z(i,j)=0$. The picture below illustrates how an obstacle is stored in a fixed grid representation.

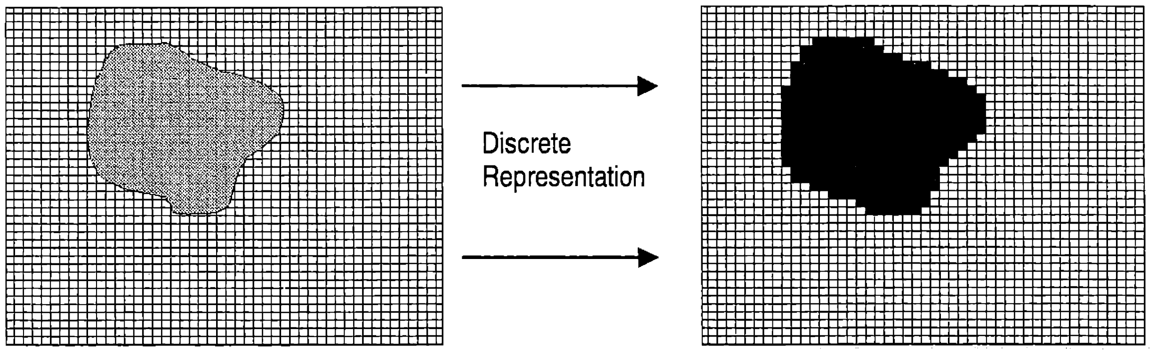


**Figure 2.1 Discrete Representation of an Obstacle**

Figure 2.1 the obstacle on the left part of the figure is discretized to the version on the right. The map on the right is represented as all 0's except for the black areas that are assigned a value of 1. Clearly, the resolution of the map, $Z_{res}$, will determine how close the discrete version maps the original obstacle.

### 2.3.3 Quadtrees

A more efficient storage of the map, can usually be accomplished through the use of quadtrees. This type of structure attempts to store data in blocks of varying size. In this manner large contiguous regions of free space can be represented as a single element in the quadtree representation [11]. A quadtree begins with the environment labeled as one large square. If an obstacle exists within the large square then the square is subdivided into four blocks of smaller size. A block is selected and is checked for an obstacle within that block. Similarly if an obstacle exists, the block is divided again into fourths. If there exists no obstacle within a block, then that block is divided no further. This logic progresses until cells of smallest grid size represent all obstacle regions. If further storage gains are desired, then grid cells labeled as obstacles can be grouped together. In order to see the storage gains, imagine a *n x n* grid of free space. A fixed grid representation will store $n^2$ zeroes, while the quadtree will story only one zero. Clearly, in sparse (low obstacle density) environments, there exists great advantage to such a structure. However, as the obstacle density increases, the advantage of such a structure decreases. In addition, the tree structure that results is more complicated to store and interpret than a fixed-gird structure.

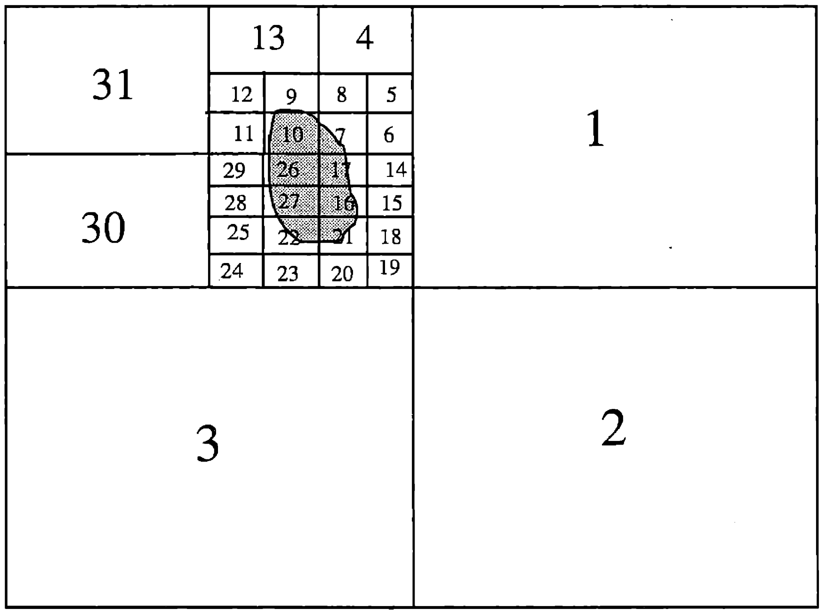


**Figure 2.2 Quadtree Representation**

Figure 2.2 shows a typical quadtree representation of a single obstacle. The space of interest is *16 x 16* grid cells. The full fixed grid representation takes 256 cells as opposed to the quadtree representation of 31 cells. The only nonzero cells in Figure 2.2 are 7, 17, 16, 21, 22 27, 26, and 10. The fixed grid representation would have the same number of nonzero cells but many more zero cells. For a more detailed review of quadtrees and other related data structures see Samet's overview on the subject [12]. Stentz, has demonstrated the use of a more complicated data structure, the framed-quadtree. This type of structure frames every block in the quadtree with cells of minimum resolution. This type of structure, while more cumbersome in storage than the normal quadtree, is specialized for path planning [17].

## 2.4 Network Representation

As previously presented in the nomenclature section a graph (or network) G is a collection of nodes and arcs that connect those nodes. Somehow the information about the environment needs to be represented as a network. Representing the environment in a graph allows for shortest path computation. Many approaches in the area of path planning bypass the network storage problem and use the representation of the environment (with some additional information) as the network. Other approaches, acquire information from the environment (or from the map) and build a separate network based on this information. Each of these two methods have their advantages and disadvantages.

### 2.4.1 Network Representation using a Fixed Grid Map

If the choice of map representation is a fixed grid, the use of this map as a network is straightforward. All that is needed is a set of nodes and arcs. First, let each grid cell represent a distinct node. Therefore, if the environment is represented as *n x n* grid cells, there are *n x n* nodes in the network. Now, the arcs need to be identified. Let two nodes *i* and *j* be neighbors if an arc $(i,j) \in A$. Therefore, $neighbor(x)$ is defined as the set of nodes reachable from node x along a single arc emanating form node *x*. Let $|neighbor(x)|$ be defined as the cardinality of the set $neighbor(x)$. For simplicity, let $|neighbor(x)|$ equal a constant for any interior node *x*. An interior node is a node in the network whose corresponding location in the graph is sufficiently far from the boundary of the map such that a single arc emanating from node x will not cross the boundary of the map. Therefore, the size of $neighbor(x)$ needs to be determined before the arcs in *G* can be defined. The simplest case is $|neighbor(x)| = 4$. In this case, 4 arcs emanate from every node in the network. In the terms of the map, 4 moves (left, right, up, and down) are possible from every grid cell. If $|neighbor(x)| = 8$, then diagonal moves are allowed. Given $|neighbor(x)| = 8$, all cells adjacent to the cell corresponding to node *x* are reachable along one arc. If $|neighbor(x)| = 16$, the neighbor structure for node *x* is illustrated in the following figure.

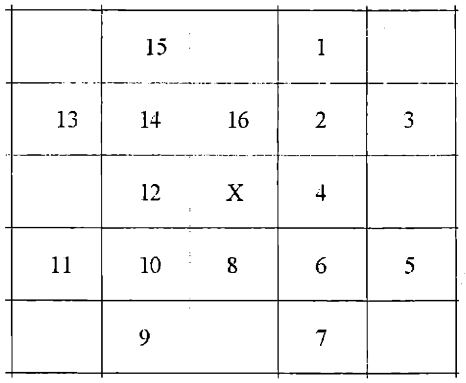


**Figure 2.3 Neighborhood Structure**

All numbered cells/nodes are elements of *neighbor(x).* These sixteen cells are chosen, because they uniquely define all possible moves to any cell that is within 2 cells of node *x*. In reference to Figure 2.3, suppose a move is desired to the non-numbered cell immediately to the left of cell #12. Adding this cell to neighbor(x) is frivolous, as moving to that cell corresponds to moving to cell #12, redefining cell #12 as node *x*, computing the new set (*neighbor(x)*), and moving to the new cell #12. In general, $|neighbor(x)| = 2^{d+2}$, where $d \in [2,\ldots,n]$ and is defined as the desired distance (in cells) from *x*. For example, if one chooses to include moves of length 3 cells, $|neighbor(x)| = 32$. Establishing the neighborhood in this manner, identifies all arcs emanating from all interior nodes in *G*. Identifying, arcs emanating from non-interior nodes, translates to using the same construction as above, but eliminating those arcs that cross the boundary.

Identifying the costs of each arc is also needed for the network representation. In general, if the transition from node *i* to node *j* is feasible a cost, $c_{ij}$ exists that is a measure of the cost of that transition. A transition from node *i* to node *j* is feasible if the transition does not cross into an obstacle, and the transition is within the modeled dynamics of the vehicle. If node j is an obstacle, assign $c_{ij} = M$, where *M* is larger than any feasible path in the graph. This construction will eliminate the consideration of this arc *(i,j)* in the shortest path. Another approach would be to eliminate node *j* from the neighborhood of node *i*. Each method is acceptable. How the cost is quantified, depends on the application. In this thesis, the euclidean distance metric is the cost of choice. However, one may wish to use transition time, vehicle safety, or any of a host of other cost metrics. Different metrics may result in different types of paths. If transition time is chosen, the resulting path is the shortest time path. If vehicle safety is chosen as the metric, the resulting path is the safest path from start to goal. Cost metrics that combine time and safety could be chosen, and the resulting shortest path is optimal in the manner that time and safety were combined in the cost metric.

On 2-D and 3-D graphs, using the euclidean distance metric produces shortest euclidean paths, constrained by the movements as identified by the neighborhood structure. For any node $X \in N$, if all other nodes *Y* are elements of *neighbor(X)*, then the shortest path produced is the shortest euclidean path. However, choosing such a neighborhood structure, creates $O(n^4)$ arcs in *G*. Constraining, the neighborhood size to 16, results in $O(n^2)$ arcs in *G*. As will be discussed later many shortest path algorithms search all arcs emanating from a node. Therefore, keeping the neighborhood size sufficiently small results in paths that may be longer than the actual shortest euclidean paths. However, this tradeoff is acceptable in terms of the decreased computational complexity.

### 2.4.2 Extensions to Fixed Grid Representation

Many times conditions arise in implementing shortest path algorithms that warrant better representations of the vehicle in the network. The cost of an arc may rely on more than just the distance between two locations. For instance, the cost of an arc may rely on the state and/or configuration of the vehicle. If one assumes the speed of the vehicle is constant, and the vehicle is operating in the plane, a vector of position and heading can represent the state of the vehicle. Other state variables may exist, but the vector composed of position *(x,y)* and heading, $\theta$, is frequently sufficient for many applications (i.e. the case where a vehicle cannot follow a jagged path). If $\theta$ is discretized, the network is now a 3-D grid. A node *i* now represents location and heading. In this setting, the cost metric chosen will now rely more closely on the dynamics of the vehicle. The length of an arc *(i,j)* and the location of that arc in space, is largely dependent on the heading at node *i* and at node *j*. Checking the validity of an arc *(i,j)* now becomes more important. For instance, an arc *(i,j)* may appear feasible, but due to vehicle dynamics the arc *(i,j)* may cross into an obstacle. Therefore, before any path computations are performed, invalid arcs will need to be eliminated.

Suppose $\theta$ has *36* discrete levels (i.e. $\theta \in \{0,10,20,30\ldots360\}$). Therefore, at any cell location *(x,y),* there exists *36* nodes in *G.* Now if the network is connected through the neighborhood structure previously defined ($|neighbor(x)| = 16$), and all levels of $\theta$ are reachable from any other level, each node in *G* will have *576 (16*36)* arcs emanating from itself. Bear in mind that increasing the dimension of the grid increased the number of nodes from $n^2$ to $36n^2$ (where the planar grid is $n$ x $n$ cells). Assuming the environment is free of obstacles, the original 2-D network had approximately $16n^2$ arcs. The new 3-D graph has approximately $20736n^2$ arcs ($576*36n^2$). The only manageable way to deal with such a growth in the number of arcs is to limit the number of levels in $\theta$ reachable from a fixed heading. However, limiting the transitions, does nothing to address the increase in the number of nodes in *G*. As can be seen, the explosion in the size of *G*, as the dimension of the graph grows, severely increases the size of the problem. For this reason, the remainder of this thesis will attempt to utilize creative methods of planning with a 2-D map and graph. The methods must be creative due to the nature of the vehicle, and the types of paths produced by a shortest path algorithm in a 2-D grid.

## 2.5 Algorithms – Approaches, Descriptions, and Performance

Many algorithms exist for finding a shortest path in a network. Since the shortest path problem is so widespread in its use, many attempts are made to speed up current algorithms with advanced data structures. The purpose of this thesis is not to discuss in great detail the advantage of using a radix-heap, as opposed to a binary-heap for the search. Different algorithms for finding shortest paths in a network may lend themselves to different data structures. Rather than present a list of algorithm performance using different data structures, an overview of the workings of different algorithms will be provided. In addition, insight as to how different shortest path algorithms operate in the network as previously defined will be given.

Some algorithms find paths from source to goal (in that order), while others search the network backward from goal to source. Most algorithms can be implemented both ways, however in the ensuing description it should be noted that some algorithms are be presented form source to goal, while others are presented form goal to source.

### 2.5.1 Greedy

Instead of focusing on the global cost objective greedy algorithms typically focus on minimizing some local cost. In the context of shortest paths, a couple of definitions are needed to motivate a greedy type algorithm.

1. *h(x):=* heuristic that estimates cost of path from node *x* to node *t* (target or goal). *h(x)* is an acceptable heuristic if it does not overestimate the cost of the optimal path from node *x* to node *t*.

2. Furthermore, let *h(x):=*Euclidean distance from node *x* to node *t* (any acceptable heuristic could be chosen. In the context of the network as previously defined, this heuristic is acceptable).

With this definition of the heuristic, the greedy algorithm follows.

```
algorithm Greedy
begin
  $s$ = current node (source);
  while $s \neq t$ (target) do
    Move to node $j$ such that $h(j)$ is minimum $j \in neighbor(s)$;
    $s = j$;
  end
end
```

**Figure 2.4 Greedy Algorithm**

The algorithm as shown in Figure 2.4 is not assured of converging on the target node. For example, in the presence of obstacles the vehicle may cycle between two or more nodes indefinitely. See the figure below for an example of this behavior.

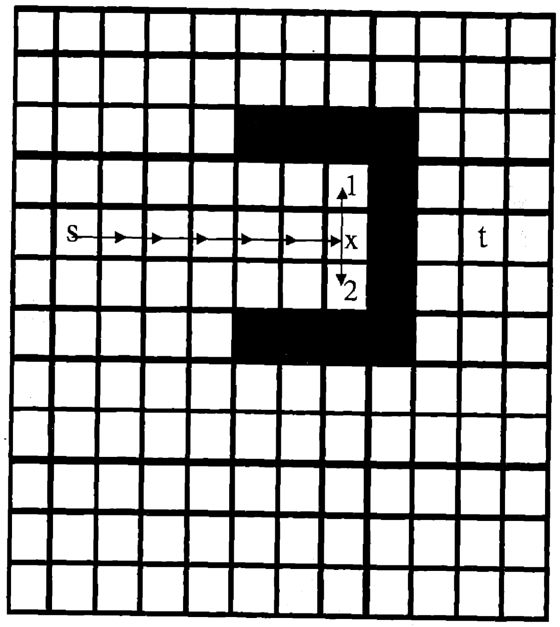


**Figure 2.5 Greedy Algorithm Getting Stuck**

In this illustration, the algorithm proceeds from s to node x at which point it chooses either node 1 or node 2. Once, the current node is set to 1 or 2, the algorithm will then set node $x$ as current node. This looping behavior will continue indefinitely.

Clearly such an algorithm is not acceptable for the problem at hand. However, if the only knowledge of the network at a current node is the only the set of neighboring nodes, then a greedy algorithm is the best one could hope to implement.

### 2.5.2 Dijkstra's

The classic approach to the shortest path problem is Dijkstra's algorithm. Dijkstra's algorithm first appeared in 1959 in [4]. This algorithm is probably the most widely used for shortest path computation.

This algorithm is capable of finding the shortest path from source to all other nodes in $G$, or in the reverse implementation, the shortest path from target to all other nodes. If all paths are not desired, the algorithm is easily terminated when the target is reached. The algorithm essentially, maintains a list, $d(i)$, which is an upper bound on the distance from source to node $i$. As the algorithm searches from the source, some nodes are labeled permanent and others are labeled temporary. The distance labels for the permanent nodes cannot be decreased. The nodes labeled temporary have distance labels that are only an upper bound. The algorithm begins at the source (or at the target in reverse implementation) and begins to label nodes in the temporary set in the order of smallest distance labels. It chooses a node with minimum temporary label, labels that node permanent, and scans all arcs emanating from that node to update other distance labels. As the algorithm progresses it decreases the distance label until no further decreases are possible. Once a node is labeled permanent, the predecessor of the permanent node can be stored. Assuming the algorithm is allowed to find shortest paths from source to all other nodes, then the shortest path to node $x$ is recovered by following the set of predecessors (sometimes referred to as backpointers) from node $x$ back to the source. The formal pseudo-code [7] is provided in Figure 2.6. As presented, shortest paths from source to all other nodes are found.

```
algorithm Dijkstra;
begin
  $S := empty; (S = permanently\ labled\ nodes)$
  $\bar{S} := N; (\bar{S} = all\ nodes\ not\ permanently\ labeled)$
  $d(i) := \infty\ \forall i \in N;$
  $d(s) := 0\ and\ backpointer(s) := 0;$
  while $|S| < n$ do
  begin
    let $i \in \bar{S}$ be a node for which $d(i) = min\{d(j)) : j \in \bar{S}\}$;
    $S := S \cup node$ i;
    $\bar{S} := \bar{S} - node$ i;
    for each $arc(i, j) \in A(i)$ do
      if $d(j) > d(i) + c_{ij}$ then $d(j) := d(i) + c_{ij}$ and $pred(j) := i;$
  end;
end;
```

**Figure 2.6 Dijkstra's Algorithm**

If the algorithm is terminated once the target is reached, at a minimum the path from source to target exists. If the set of permanently labeled nodes contains more than just the target, then the shortest paths to all nodes in the set of permanently labeled nodes also exists. The information that exists in the collection of backpointers will later be shown to be important in the development of tracking a moving target.

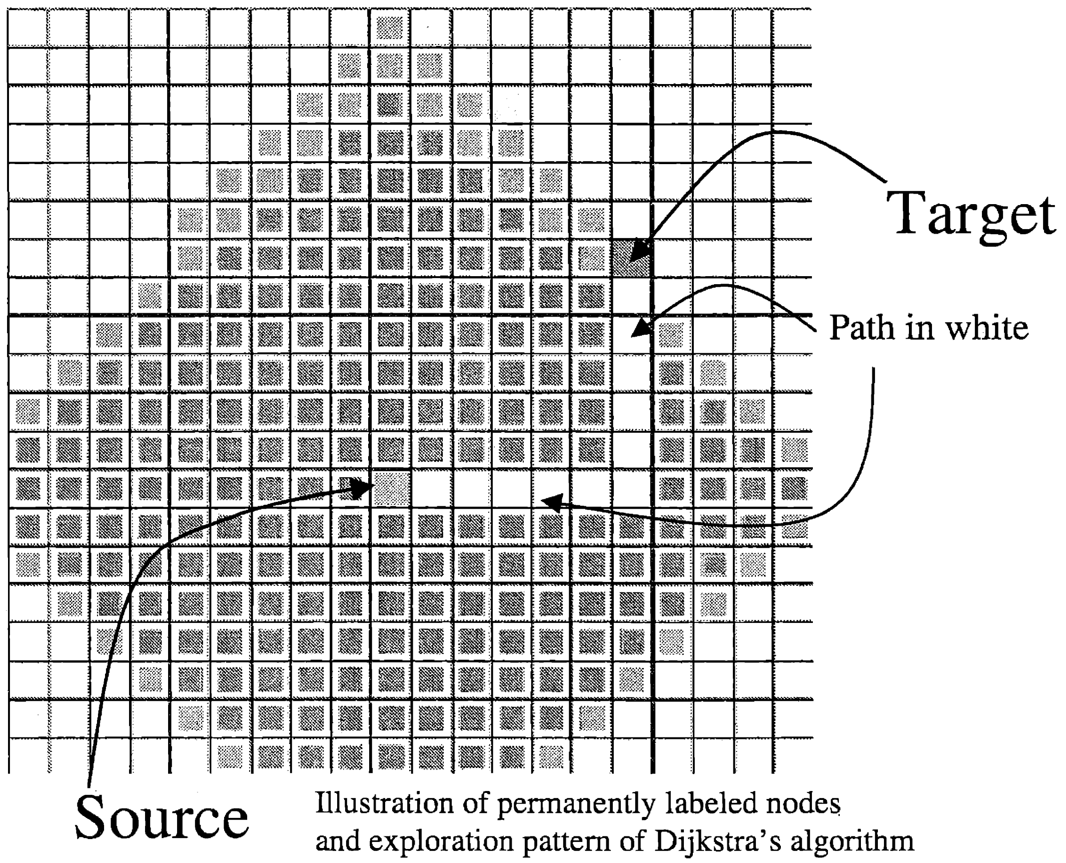


Illustration of permanently labeled nodes and exploration pattern of Dijkstra's algorithm where |neighbor(x)| =4.

**Figure 2.7 Dijkstra's Search Pattern**

Figure 2.7 illustrates how Dijkstra's algorithm fans from the source to all other nodes in its search for the path to the target. In Figure 2.7 all shaded nodes are in the set of permanently labeled nodes. Thereby, if the backpointers at each node are stored, a unique path exists from the permanently labeled nodes to the source. With neighborhood structures as previously defined, the set of permanently labeled nodes will tend to fan out from source (or target in reverse implementation) as seen in Figure 2.7.

Various implementations of Dijkstra's algorithm have been developed through the years, in attempts to obtain better performance. The generic algorithm runs in $O(n^2)$. Depending on the implementation chosen, the best time bound is $O(\min\{m+n\log n,\ m\log\log C,\ m+n\sqrt{\log C}\})$, ($C$ is a constant satisfying certain properties specific to each algorithm). The details of the complexity arguments are not included, but can be found in [7].

### 2.5.3 A*

A* search is a slight modification to Dijkstra's search that performs extremely well on grid type networks. Pearl [9] provides a detailed description of A* search. A* is able to utilize a heuristic estimate of the length of the path at each node, in an attempt to eliminate needless searching of the graph. If the heuristic estimate is a lower bound on the actual path cost, then an optimal path will result from the A* algorithm. In 2-D grid networks, an acceptable heuristic is the Euclidean distance to goal,

$$\begin{aligned} &h(i)=\sqrt{(x_i - x_{goal})^2 + (y_i - y_{goal})^2}, \\ &\text{where} \quad x_i := x \text{ coordinate of node}(i), \\ &\qquad\qquad y_i := y \text{ coordinate of node}(i). \end{aligned} \tag{2-3}$$

The A* algorithm can be thought of as nothing more than Dijkstra's algorithm with a different set of costs, $c_{ij}^h$, instead of $c_{ij}$. This transformation applies when the heuristic does not overestimate the distance to goal.

$$c_{ij}^h = c_{ij} - h(i) + h(j) \quad \forall (i,j) \in A \tag{2-4}$$

The following figure shows the search savings using A*. The search savings are clear when comparing this figure with Figure 2.7

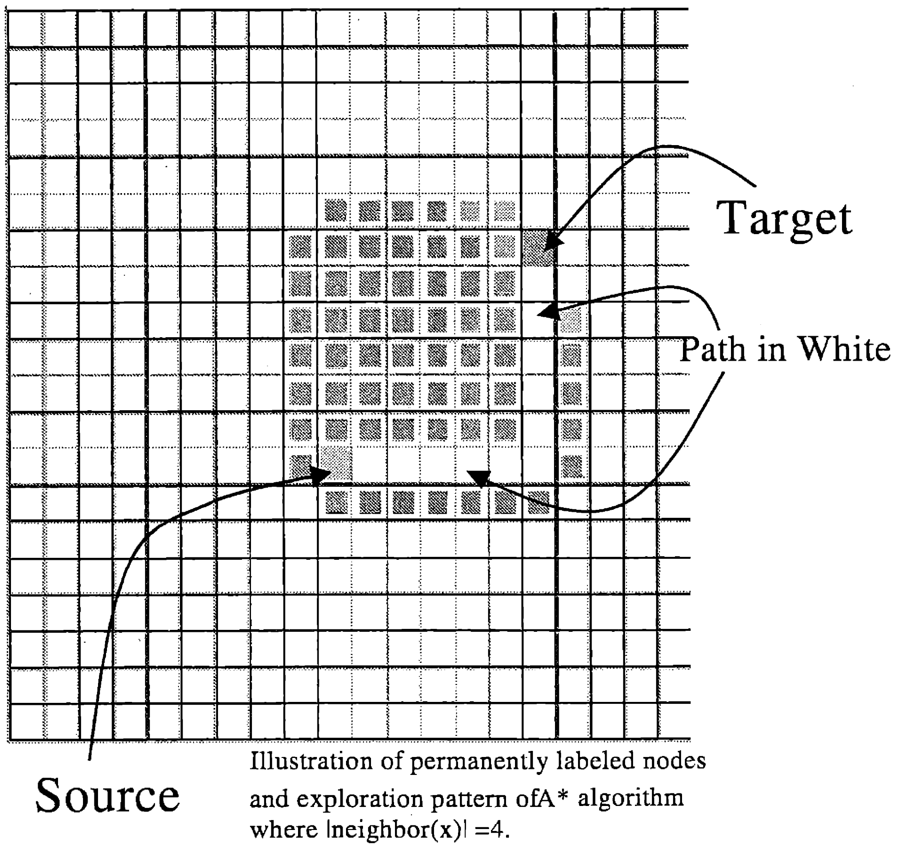


Illustration of permanently labeled nodes and exploration pattern ofA* algorithm where |neighbor(x)| =4.

**Figure 2.8 A* Search Pattern**

The worst case performance of a shortest path algorithm using the transformed costs as dictated by A*, is still the worst case performance of the algorithm without the transformed costs. Since the heuristic is only a lower bound on the actual path cost, an acceptable heuristic is 0. In this case the heuristic has provided no information and the A* algorithm has provided no advantage. However, as illustrated in Figure 2.8 the typical performance (especially on grid-based networks) of A* should be much better than the typical performance of a generic shortest path algorithm.

### 2.5.4 D*

The previous two algorithms are designed to work well on static networks. Stentz in [14], [15], and [16], has developed an algorithm that is capable of performing shortest path computations on dynamic networks. In general, this algorithm is capable of handling changing arc costs while the problem solver is traversing the graph towards the target. The costs may change as the vehicle (or problem solver) discovers more accurate information about its environment. If the vehicle is tasked with sensing its environment, it may discover new obstacles en-route to the target. These new obstacles can then be represented by new arc costs (per previous discussion on infinite or large arc costs). Unlike some dynamic shortest path problems, frequently seen in the literature, this algorithm assumes no structure in the way arc costs may change. The ability to handle the dynamic network in this manner is paramount to the performance of the vehicle's path planer. The details of this algorithm are given in Appendix A. It should be noted that in the static case this algorithm is essentially Dijkstra's algorithm. Therefore, the running time for this algorithm in the static case will be $O(n^2)$. In his summary, Stentz notes that D* "is able to handle any path cost optimization problem where the cost parameters change during the search for the solution" [15]. Furthermore, he states "that the algorithm is most efficient when changes occur near the starting point in the search space, which is the case with a robot (UAV) equipped with an on-board sensor". The pseudo-code for this algorithm is provided in Appendix A. However, for more information see [14], [15], and [16].

### 2.5.5 Moving Targets and Shortest Path Algorithms

This section provides a quick overview of the moving target problem and how the previous algorithms can be configured to work in this scenario. Generally, moving the target is a bad thing to do if one wants to recompute a new shortest path. Simply, the cost from any node to the target is dependent on where the target is located. Changing the location of the target will invariably change much of the data, and variables in a shortest path problem.

The greedy algorithm will suffer the least in terms of performance for the moving target problem. The greedy algorithm only relies on local information to find a path. However, the problems inherent in the greedy algorithm, as previously explained, still exist in the moving target scenario.

When the general Dijkstra's or A* algorithm is used in the moving target scenario, a full replan is needed to compute a path to the new position of the target. A full replan is defined as reinitializing the algorithm with new source and target information, and then performing the algorithm to find the new shortest path. Frequently, performing the full replan is not feasible for a vehicle wishing to track a moving target. The D* algorithm may perform better than performing a full replan, but noting the warning of changes to the target, it is doubtful that the D* algorithm will compute new paths as efficient as one may need.

Using Dijkstra's or A*, to perform a full replan, will have the same worst case bound as the static shortest path algorithm. If quick computation of paths is essential to convergence on the moving target, performing full replans may not yield run-times of acceptable length. In the case of grid maps and sparse obstacles, the number of cells or nodes permanently labeled tends to be a function of the distance from source to target. In addition, the running time of the algorithm is directly related to the number of permanently labeled cells. This statement is clearly illustrated in the no obstacle case as shown in Figure 2.7. If the vehicle begins far away from the target, recomputing paths to a moving target while advancing along those paths may cause the vehicle to wait while the shortest path computations complete. If the target is moving through the environment, limited by its maximum speed, it may be more beneficial for the seeker to travel towards the target rather then along an optimal cost path. The reason for this benefit may lie in the assumption that traveling towards the target is less computationally expensive than travelling along an optimal path. This concept is illustrated in the following argument. Assume the vehicle is travelling along a path computed with the most recent target location. If the target location is changed and a new path is computed, the previous path may still do a good job at getting the vehicle close to the target (assuming the target locations did not change too drastically). This method, expends no computation in moving towards the target. Therefore, the vehicle could sacrifice the optimal solution and just travel along the old path. However, if this logic continues, the seeking vehicle will never track the moving target. If paths could be efficiently maintained from goal to goal, then once the original goal is attained, the seeking vehicle could travel along the paths computed from goal to goal. This idea is the motivation for the moving target path planner as later described in Chapter 5.

## 2.6 Shortest Path Algorithms on a UAV

Implementing a shortest path algorithm on a UAV, can result in numerous difficulties. UAVs are typically faced with performing complex tasks in spite of their limited computing power and memory. Therefore, a computationally cheap and robust path planner is essential to the success of a UAV. However, most important is the marriage between the shortest path code and the trajectory generation code. Once a shortest path is found, a trajectory must be found that follows that path. The inability to follow the path may lead to unsafe travel for a UAV.

One issue to always consider in implementing path planning on-board a UAV, is the dynamics of the vehicle. In a perfect world, a vehicle could follow any path produced by the shortest path algorithm. If a vehicle operates in the plane, typically, paths produced by shortest path algorithms (on grid based networks) are jagged (not differentiable). Most vehicles cannot follow such a path, and must generate a trajectory that is within its dynamics and closely approximates the shortest path. Following an approximating trajectory, subjects the vehicle to increased likelihood of obstacle collision. If an approximated trajectory is followed, assuring the safety of the trajectory is essential to the vehicle's safety.

The methods as outlined in Section 2.4.2 (i.e. using the heading in the planning) will produce paths that are smooth and admissible in terms of the vehicle dynamics. However, as previously stated, computing paths with this type of network can be computationally burdensome. Therefore, the remainder of this thesis attempts to create a planner that is capable of operating on a 2-D planar grid

# Chapter 3

# Path Smoothing

The path returned by a shortest path algorithm is composed of a collection of arcs and nodes. In the context of a 2-D grid map, each node refers to the center of a grid cell and an arc refers to the line segment connecting two adjacent grid cells. The resulting path is jagged and cannot be accomplished by most UAVs. Specifically, most UAVs cannot experience instantaneous changes in velocity as implied by the jagged portions of a path. Some vehicles may be able to stop at a jagged portion of the path, and then align there heading with the desired heading of the path. If this maneuver is possible, then the instantaneous change in velocity does not have to hold. However, such a maneuver is generally not desired or feasible for many vehicles. Therefore, some type of path smoothing must be employed in order for the ground track to be realized by the vehicle. However, smoothing the path may result in a smoothed trajectory that is no longer safe. Safety as applied to a trajectory, refers to a trajectory that does not hit or cross into an obstacle. The only assurance the vehicle has of safety is to stay on the computed shortest path. Any deviation from this path may compromise the safety of the vehicle. Subsequently, the deviation from the jagged shortest path to the smoothed trajectory is a measure of safety and should be minimized. Techniques in the map building (described in Chapter 4) phase can be used to provide margins of safety for the vehicle, such that a certain level of deviation is acceptable and safe.

## 3.1 Definitions and Introduction

Let a certain path, $p$, be defined as a set of n Cartesian coordinates $(x, y)$ and the linear segments that connect point $j$ to point $j+1$. Now, let the vector, $\boldsymbol{X}$, represent the collection of data that corresponds to the $x$ data points, and the vector, $\boldsymbol{Y}$, to correspond to the $y$ data points. Let $\boldsymbol{X(i)}$ refer to the $i^{th}$ element in the vector $\boldsymbol{X}$, $\forall\ i = 1,\ldots,n$.

Likewise, let ***Y(i)*** refer to the $i^{th}$ element in the vector ***Y***, $\forall\, i = 1,...,n$. Furthermore, let $P(i) = (X(i), Y(i))$ and ***P = [X,Y]***. Figure 3.1 refers to a possible path with the 4 data points represented as asterisks, *.

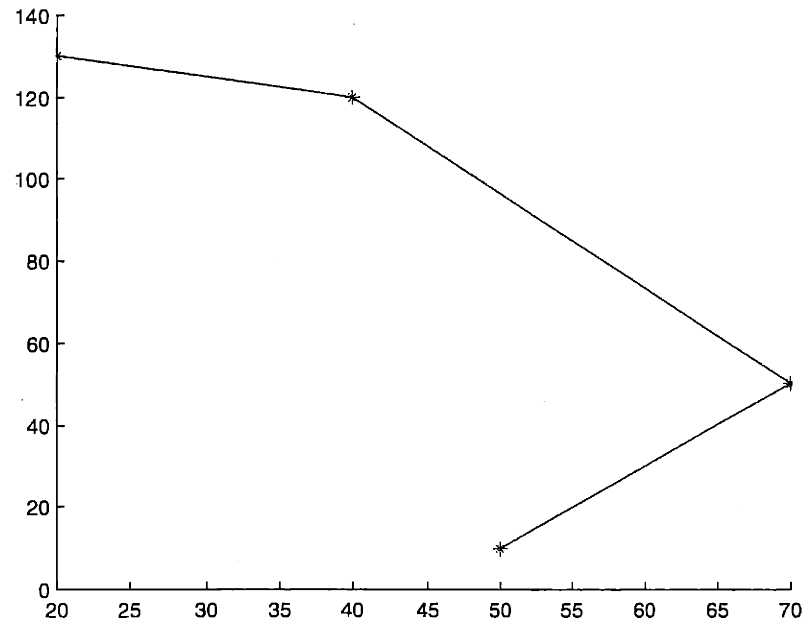


**Figure 3.1 Jagged Shortest Path**

Notice that the path, *p,* as shown above, is jagged and non-functional. Here non-functional refers to a path that is not a function (i.e. *y* is not a function of *x*). A path in the 2-D space will never, in general, behave functionally. In the context of a UAV, a trajectory needs to be created that eliminates the jagged nature of *p* and does not deviate too far from *p.* The need for a non-jagged trajectory is necessitated by the inability of a vehicle to follow a jagged path. A smoothed path needs to be generated that is within the vehicle's capabilities and does not deviate too far from *p*.

Given that *p* is non-functional, normal smoothing techniques of finding some function *f,* such that $y \approx f(x)$, will not work. Several approaches to this problem will be described with the main emphasis placed on conventional techniques of polynomial data smoothing. As implemented, these methods rely on the least-squares formulations with linear equality and inequality constraints. The use of the least-squares solution for an over-determined linear system is advantageous due to the closed form solution (i.e. low computational expense). However, these techniques are not able to explicitly handle the dynamics of the vehicle. In addition, the jaggedness of the path is not directly handled. However, the jaggedness can be eliminated as a by-product of the functions chosen. Other optimization formulations that are more apt to handle some of the problem variations discussed in Section 3.2 can be used. However, these formulations will rely on more general optimization techniques and will be discussed in Section 3.4. These methods, although more complete in their formulation, will suffer from computational expense and may converge to local optimum solutions.

Other methods that rely on techniques from optimal control are able to incorporate the dynamics of the vehicle but suffer from heavy computational burden. These methods are able to constrain certain parameters of the dynamic model (e.g. rudder deflection, yaw rate, dive rate, etc.) to lie within a certain feasible range. In addition, another nice feature of these methods is the ability to constrain the state of the vehicle at any time through the trajectory of the vehicle. Typically these optimal control methods are employed in off-line applications such as generating trajectories for a rocket before the launch occurs. These methods are typically quite computationally burdensome and as such are not investigating further. However, as computing power increases these methods may warrant further exploration.

Finally, a hybrid approach is developed. This approach uses the least squares formulation, with or without the linear constraints, and blends the result with another method that is capable of handling a certain type of problem variation. This approach relies on geometric arguments and properties of the vehicle's dynamics. This method is the one of choice and is implemented within the larger simulation.

## 3.2 Problem Variations

In the simplest case, the objective is to generate a smoothed trajectory that closely follows *p*. For the simple case, no explicit constraints to the path are provided. However, variants to the problem may necessitate placing constraints upon the smoothed trajectory. These variants may include constraining the smoothed path to certain points (e.g. setting the start and end points). In the context of smoothing a trajectory, the start position usually must be fixed to the position of the vehicle. Likewise, it may be important that the smoothed trajectory completes itself at en exact desired location. Let us call these two conditions the START-END constraints. Furthermore, the curvature of the trajectory may be constrained to lie within certain bounds (the curvature constraint is herein labeled the CURVATURE constraint). Curvature is a measure of how sharply a trajectory turns. Finally, specifying the initial heading of the vehicle (i.e. slope of the trajectory) at the start point may be an essential modification to the simple case. The reason for this constraint lies in piecing one smoothed trajectory with another. When piecing two trajectories together one knows that the start point for the second trajectory is the end point for the first trajectory. Similarly, the initial heading of the second trajectory is the end heading of the first trajectory. This restriction is heretofore referred to as the START-HEADING constraint.

Certain approaches to the path-smoothing problem are more capable at handling some of the constraints than are others. However, the ability to formulate a constraint with the specified approach may not translate to adequate solutions or acceptable algorithmic performance.

## 3.3 Polynomial Data Fitting (Unconstrained)

For the remainder of this section, let us only address the problem of generating a smooth trajectory that closely matches the path *p* and not the problem of explicitly creating a trajectory that is within the vehicle's dynamics. Hopefully, the path created by trying to closely match *p* will be within the capabilities of the vehicle.

### 3.3.1 Definitions and Development

Assume that a vector ***S*** exists, such that a mapping, *f*, from $S(i)$ to $X(i)$, and a mapping, *g*, from $S(i)$ to $Y(i)$ $\forall i = 1,\ldots,n$ exists. Thereby, two functions *f* and *g* could be used to estimate any $(x,y)$ pair. Specifically, $X(i)$ is approximated as, $X_{est}(i) = f(S(i))$, *and* $Y(i)$ is approximated as $Y_{est}(i) = g(S(i))$ $\forall i = 1,\ldots,n$. Thus, for a given value of $S(i)$ one can recover $P(i)$ through the functions *f* and *g*.

Two questions remain. How does one find a proper value of *S(i)* to associate with each data point, and how does one find *f* and *g*? The most intuitive method (in the context of a vehicle) for estimating ***S*** would be to let *S(i)* equal to the time it would take the vehicle to traverse along the smooth path to the point *[f(S(i)),g(S(i))]*. However, finding this time variable is highly dependent on *f* and *g*. Since the independent variable is hopefully created as data, using time will not be acceptable.

Another choice for ***S***, is the cumulative arc length along the jagged path. If the mapping of ***S*** through *f* and *g* results in a trajectory exactly the same as the jagged path, then ***S*** is also the cumulative arc length of the trajectory. The closer the smooth trajectory is to the jagged path, the better ***S*** estimates the actual arc length of the smooth trajectory. If the magnitude of the vehicle's relative velocity (i.e. relative to wind or current), and the velocity of the current (or wind) are constant, then ***S*** is also an approximation of the trajectory time. Since time is unavailable as an independent variable, let us estimate the length of the arc from *[X(1),Y(1)] to [X(i),Y(i)]* $\forall$ *i=1,…,n* and use this arc-length estimate as ***S*** (see [1]). The use of the estimated arc length should prove successful in light of velocity assumptions.

$$\begin{aligned} &S(1)=0 \\ &S(i)=S(i-1)+\sqrt{\left(x(i)-x(i-1)\right)^2+\left(y(i)-y(i-1)\right)^2} \quad \forall i=2,3,\ldots,n \end{aligned} \tag{3-1}$$

Now define vector valued function $r(s)=\langle f(s),g(s)\rangle$, where s$\in$ *[S(1),…,S(n)]*. The first element of the vector, *r(s),* corresponds to the *x* variable and the second element refers to the *y* variable. This vector-valued function corresponds to the smoothed path. A specific value for *s* will correspond to a location along the smoothed path. This definition of the vector-valued function is needed in later sections.

How does one estimate the functions, *f* and *g*? First, a measure of how well a function fits the given data must be chosen. The most widely used method focuses in minimizing the sum of the squared errors. Specifically construct the problem in the following manner

$$\min \sum_{i=1}^{n}\left[x(i)-f\left(s(i)\right)\right]^2 \quad \forall i=1,\ldots,n \quad and$$
$$\min \sum_{i=1}^{n}\left[y(i)-g\left(s(i)\right)\right]^2 \quad \forall i=1,\ldots,n \tag{3-2}$$

If a weight on each data point is desired, such that the squared error at a certain point is weighted differently than at other points, a vector of the weights, *w(i) ∀ i = 1,...,n* is created. The formulation now becomes

$$\min \sum_{i=1}^{n} w(i)*\left[x(i)-f\left(s(i)\right)\right]^2 \quad \forall i=1,\ldots,n \quad and$$
$$\min \sum_{i=1}^{n} w(i)*\left[y(i)-g\left(s(i)\right)\right]^2 \quad \forall i=1,\ldots,n \tag{3-3}$$

Another formulation, which transforms the problem into a linear optimization problem, minimizes the sum of the absolute errors. The transformation involves an introduction of new variables, and is given in [3].

$$\min \sum_{i=1}^{n}\left|x(i)-f\left(s(i)\right)\right| \quad \forall i=1,\ldots,n \quad and$$
$$\min \sum_{i=1}^{n}\left|y(i)-g\left(s(i)\right)\right| \quad \forall i=1,\ldots,n \tag{3-4}$$

Finally, another popular technique minimizes the maximum errors. This model can also be represented as a linear optimization problem, through a similar transformation. The weights, *w(i)*, can also be applied to this formulation.

Given these three optimization models, the forms of the functions are still left to be identified. For the remainder of this section, the formulations as seen in (3-3) will be used. This formulation is probably the most widely used, and often has a closed form solution. If the class of $p^{th}$ order polynomials is chosen to fit the data, the formulation will look as follows for the x data:

$$\min \sum_{i=1}^{n} w(i)\left[x(i)-\left(\alpha_0+\alpha_1 s(i)+\alpha_2 s(i)^2+\alpha_3 s(i)^3+\ldots+\alpha_p s(i)^p\right)\right]^2 \quad \forall i=1,\ldots,n \tag{3-5}$$

and for the y data:

$$\min \sum_{i=1}^{n} w(i)\left[ y(i) - \left( \beta_0 + \beta_1 s(i) + \beta_2 s(i)^2 + \beta_3 s(i)^3 + \ldots + \beta_p s(i)^p \right) \right]^2 \quad \forall i = 1, \ldots, n \tag{3-6}$$

How does one estimate *α = (α₀, α₁,...,αₚ)* (the same will apply to *β*)? *α* has a closed form solution, as follows:

$$\vec{\alpha} = \left[ A^T W A \right]^{-1} A^T \vec{x}$$

$$where \ \ A = \begin{bmatrix} 1 & s(1) & s(1)^2 & \ldots & s(1)^p \\ 1 & s(2) & s(2)^2 & \ldots & s(2)^p \\ 1 & s(3) & s(3)^2 & \ldots & s(3)^p \\ 1 & \vdots & & & \vdots \\ 1 & s(n) & s(n)^2 & & s(n)^p \end{bmatrix},$$

$$W = \begin{bmatrix} w(1) & 0 & 0 & 0 & 0 \\ 0 & w(2) & 0 & 0 & 0 \\ 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & 0 & w(n-1) & 0 \\ 0 & 0 & 0 & 0 & w(n) \end{bmatrix},$$

$$x = \begin{bmatrix} x(1) \\ x(2) \\ x(3) \\ \vdots \\ x(n) \end{bmatrix} \tag{3-7}$$

In the above equation *w(i)* refers to the relative weight one places on the $i^{th}$ data point. If all data points are equally important, setting *w(i) = c* (a constant*)* $\forall$ *i=1,...,n* is correct.

Many software packages have the capability of efficiently performing the above matrix and vector manipulations without explicitly creating many of the above matrices as shown in (3-7).

In the example shown in Figure 3.1, *n = 4.* The algorithm as it is designed will attempt to minimize the squared error only at the 4 data points. Between the data points the function will be free to vary and will tend to lie along the piecewise linear segments. Thus, extra points need to be added between the original 4 data points. The number of points to be added between the original data points should be proportional to the length of the individual segments. Using this logic, long segments will have more data points than shorter ones.

### 3.3.2 Capabilities

The least-squares approach as presented has some significant advantages, but at the same time is hampered by its inability to explicitly model any of the aforementioned problem variations. The model as described is only concerned with fitting a $p^{th}$ order polynomial to the data that comprises ***P***.

Even though the constraints cannot be explicitly defined all hope is not lost in incorporating some of these constraints into the model. The START-END constraint can be taken into account with careful adjustments of *w(i).* In the context of smoothing a vehicle's trajectory, weighting the initial point is usually extremely important. Through repeated testing, allowing $w(1) = 10{,}000$ and $w(i)=1 \quad \forall\, i = 2,\ldots,n$, produces trajectories that begin at the desired location. Depending on the importance one places on the end point, different *w(n)'s* can be specified. In the context of this thesis, setting $w(n)=1000$ has shown good performance. Utilizing the weighting vector in such a manner is similar to Big-M methods in Linear Programming [3]. In this method a constraint or set of constraints are relaxed and the amount of deviation from the constraints is penalized in the objective function. The weighting scheme essentially penalizes any solution that violates the START-END constraint.

In order to maintain the least squares solution, incorporating the CURVATURE constraint and the START-HEADING constraint is impossible. These constraints deal with combinations of the variables present in the functions, *f* and *g*, and at the least the formulation is non-separable with respect to the *x* and *y* components. However, later sections highlight the techniques used to deal with the CURVATURE and START-HEADING constraints in the context of the least-squares result.

### 3.3.3 Implementation Considerations

Upon implementation the degree of the polynomial, $p$, must be specified. In practice, when dealing with realistic trajectories, setting $p=10$ through $p=15$ has led to good results. Setting the degree of the polynomial too low yields fits that do not match the desired trajectory. Setting the polynomial degree too high (i.e. setting $p$ close to the number of data points) will yield a fit that is only good at the data points and poor between the data points. In addition, the number of data points used has an important affect on the performance of the technique. Repeated analysis has shown that setting the number of points for each linear segments at least equal to eight and keeping the ratio of points from one segment to another dependent on the relative length of each segment is a good choice.

A simple algorithm for generating the points is provided. First, find the segment of minimum length and equally space 8 points on this segment. In addition, put a point at each vertex point for this trajectory. Now add $q$ points for the remaining $k$ segments. For a specific segment, $i$, find the ratio of its length to the minimum length segment (ratio>1). Now, multiply this ratio by the number of points used for the minimum length segment. Rounding this result, yields the appropriate number of points that are equally spaced on segment $i$ (as before add points for the vertices and do not repeat a previously stored vertices). This method should prevent the creation of excessive points. Figure 3.2 shows the creation of the data points as described (using 2 points instead of 8). Using too many points has the drawback of requiring excessive computational time. If we assume the computation of $\alpha$ in (3-1) is an $O(n^2)$ operation, doubling the number of points will quadruple the number of steps in the computation. However, using too few points will create a smoothed path, which does not adequately meet the objective.

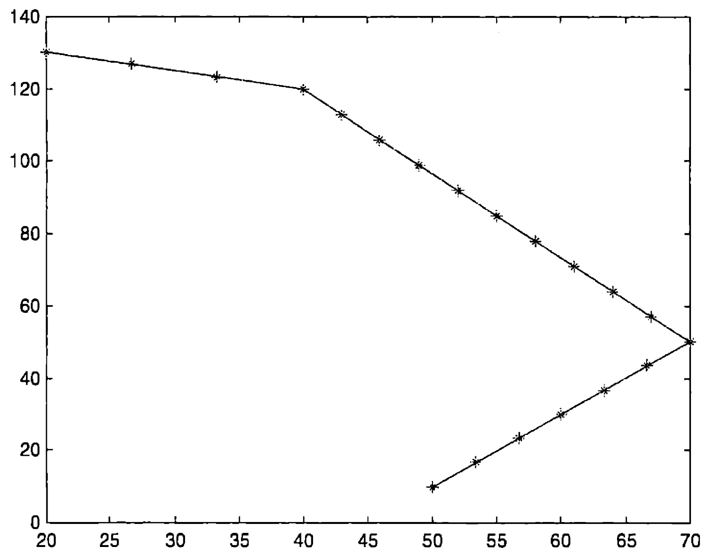


**Figure 3.2 Jagged Path with Extra Points**

Now that the vectors $X, Y$ and $W$ are specified, all that remains is to compute the least squares solution. Figure 3.3 shows both the desired path and the resulting smoothed trajectory. In this example $p = 11$.

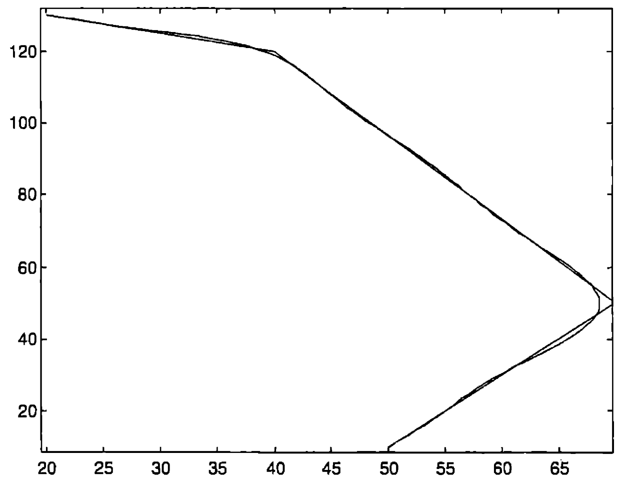


**Figure 3.3 Typical Smoothed Path**

The methods as developed to this point are able to deal with the general problem of path smoothing. No constraints have been imposed on the estimation of f and g, which are specific to the creation of a smooth path for a UAV. As seen in the section describing problem variations, the general result as seen in Figure 3.3 may be inadequate to meet specific vehicle constraints. Thereby, further discussion on implementation of this method, as developed, is premature.

## 3.4 Polynomial Fitting (Constrained)

The method as developed in Section 3.3 is unable to handle any of the problem variations explicitly in its formulation. As seen in Section 3.3.2, only the START-END constraint is effectively handled through careful adjustments to the weighting vector. One may pose the following question - *How does one implement a polynomial type data fitting in the presence of the two additional constraints, START-HEADING, and CURVATURE?* The need to include these conditions sacrifices the ability to use the least-squares solution. The new model must attempt to minimize some objective function while satisfying the three constraints (as described in Section 3.2).

### 3.4.1 Formulation

This section will introduce the formulation of the constrained optimization model. Alternate objective functions could attempt to minimize the added error for both x and y, or minimize the sum of the maximum errors. The form chosen, attempts to minimize the error norm squared. The squared error at a specific *s(i)* equals the squared error due to estimating *x(i)*, plus the squared error associated in estimating *y(i)*. Let $e(i) = [x(i) - f(s(i)), y(i) - g(s(i))]$. The following figure is a graphical representation of the error at a specific point.

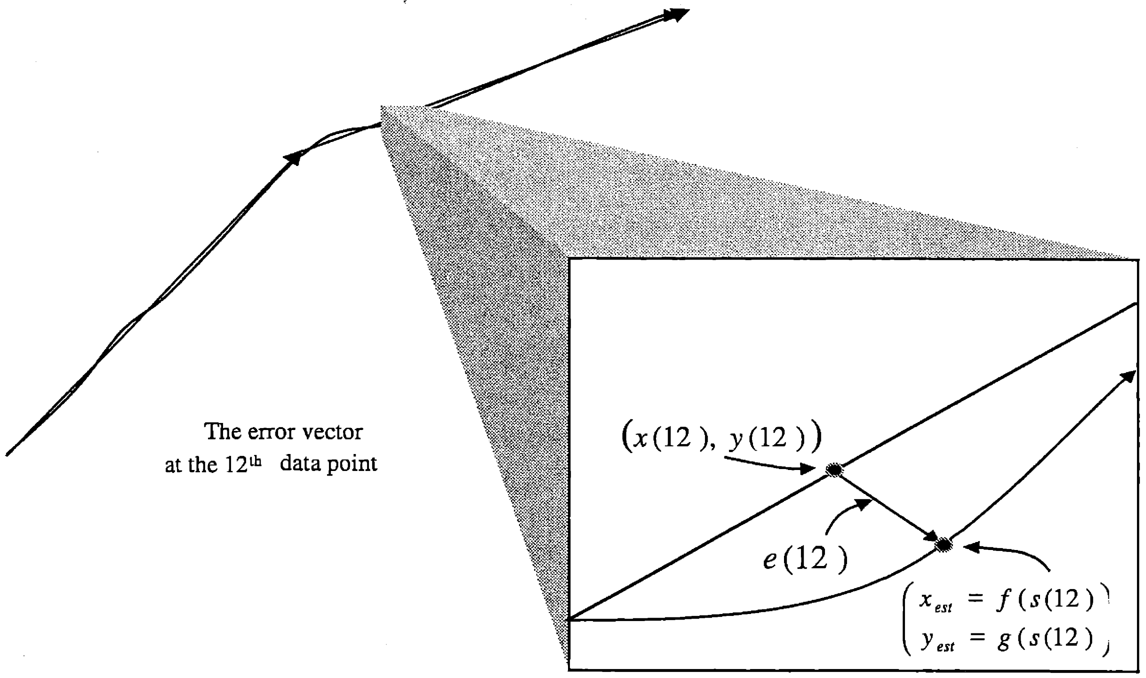

The error vector at the 12th data point
$(x(12), y(12))$
$e(12)$
$\begin{pmatrix} x_{est} = f(s(12)) \\ y_{est} = g(s(12)) \end{pmatrix}$

**Figure 3.4 Illustration of Error Vector**

A solution that minimizes $\|e(i)\|^2$, will also minimize $\|e(i)\|$. Conveniently, the objective function is written in terms of the squared magnitude, eliminating a square root. In the case where *f* and *g* are polynomials, the objective function is essentially the combination of the two functions in (3-2)

$$NLP: \min_{f,g}\left(\sum_{i=1}^{n}\left[x(i)-f(s(i))\right]^2+\sum_{i=1}^{n}\left[y(i)-g(s(i))\right]^2\right)=\min\sum_{i=1}^{n}\|e(i)\|^2 \tag{3-8}$$

*subject to:*

$$\left\{\begin{bmatrix} f(s(1)) = x_0 \\ g(s(1)) = y_0 \\ f(s(n)) = x_f \\ g(s(n)) = y_f \end{bmatrix}\right\} \tag{3-9}$$

$$\left\{K(s)_{curvature}\ \forall s\in[0,s(n)] \le \frac{1}{turn\ radius}\right\} \tag{3-10}$$

$$\left\{ \dot{r}(0) = t\left[\dot{U}_x, \dot{U}_y\right], where\, t > 0 \right\} \tag{3-11}$$

$$\left\{ \begin{array}{l} S(1) = 0 \\ S(i) = S(i-1) + \sqrt{\left(x(i) - x(i-1)\right)^2 + \left(y(i) - y(i-1)\right)^2} \\ \qquad \forall i = 2,3,\ldots,n \end{array} \right\} \tag{3-12}$$

Equations (3-9), (3-10), (3-11) represent the constraints as described in the section on problem variation. The first three constraints are the START-END, CURVATURE, and START-HEADING constraints, respectively..

In (3-11), $\left[\dot{U}_x, \dot{U}_y\right]$ is the initial unit velocity vector. The variable *t* is a new variable in the model, and must be non-negative to ensure that the initial heading vector, $\dot{r}(0)$ of the smooth path aligns with the initial velocity vector.

Constraint (3-12) is the method by which the data, ***S***, is created and is included for completeness. It should be noted that (3-10) could be discretized in the following manner:

$$2. \left\{ \max_{i=1..n} K(s(i))_{curvature} \leq \frac{1}{turn\ radius} \right\} \tag{3-13}$$

This discrete representation necessitates *n* evaluations of the curvature function, as opposed to finding the maximum curvature value over a continuous variable. Constraining the curvature at specific locations may not ensure that the curvature constraint is violated at points between those chosen locations. However, choosing enough points on the trajectory (these points denote where the curvature constraint is used) should ensure the curvature constraint is satisfied for all points along the trajectory. Similar to previous methods the objective function could be augmented by some large constant times the maximum curvature. In this way, the objective function would attempt to minimize both the squared error vector and the curvature.

The curvature function is given by,

$$K(s)=\frac{\|\dot{r}(s)\times\ddot{r}(s)\|}{\|\dot{r}(s)\|^3}\ \forall s\in[S(1),\ldots,S(n)] \tag{3-14}$$

### 3.4.2 Discussion

This optimization problem, while formulated in a straightforward manner, does not lend itself to easy solution methodologies. The variables in the aforementioned model depend on the form the functions, *f* and *g*, take. As used in previous approaches, the class of $p^{th}$ order polynomials defines the form that the functions *f* and *g* take. If our two functions are defined as seen in the least-squares model, the variables of interest are $\alpha$, and $\beta$. This NLP (non-linear program) cannot be separated into two separate problems (one problem for the *x* data, and the other for the *y* data). This non-separable property is due to the interaction of the $\alpha$, and $\beta$s in the constraints (3-10) and (3-11). After some manipulation the objective function can be rewritten as,

$$\min_{\alpha,\beta}\left\|C*\begin{bmatrix}\alpha\\\beta\end{bmatrix}-d\right\| \tag{3-15}$$

where $\alpha$, and $\beta$ are vectors of length *p*, and correspond to the coefficients of the polynomials describing *f*, and *g*. ***C*** is a matrix, with 2*n* rows and *2p* columns, and *d* is a vector of length 2*n*. The matrix ***C*** is data, and corresponds to

$$\boldsymbol{C}=\begin{bmatrix}A&0\\0&A\end{bmatrix} \tag{3-16}$$

where *A* is defined in (3-7). Some of the constraints can be rewritten in the following manner. The *d* vector is the data vector composed of the data *X,Y*.

$$d=\begin{bmatrix}X\\Y\end{bmatrix} \tag{3-17}$$

The 1st set of constraints become linear equalities,

$$\begin{Bmatrix} A_{1,1:p}\alpha = x_0 \\ A_{1,1:p}\beta = y_0 \\ A_{n,1:p}\alpha = x_f \\ A_{n,1:p}\beta = y_f \end{Bmatrix} \qquad (3\text{-}18)$$

where $A_{1,1:p}$ references the 1st row of the *A* matrix. In (3-11) ,

$$\dot{r}(0) = [\alpha_1, \beta_1] \qquad (3\text{-}19)$$

After adding the new variable *t,* three constraints must be added to the model.

$$\begin{Bmatrix} \alpha_1 = t\dot{U}_x \\ \beta_1 = t\dot{U}_y \\ t > 0 \end{Bmatrix} \qquad (3\text{-}20)$$

In general, most solvers cannot handle the strictly greater than constraint. So, one can rewrite the last constraint in (3-20) as,

$$t \geq \varepsilon, \text{where } \varepsilon \text{ is a small positive number} \qquad (3\text{-}21)$$

At this point all but the curvature constraints are represented as linear equalities or inequalities. The curvature constraint is much more difficult to formulate. The curvature function written in terms of the decision variables, $\alpha$, and $\beta$, looks as follows:

$$K(s,\alpha,\beta,p) = \frac{\left|\left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right) * \left(\sum_{j=2}^{p} (j^2 - j)\beta_j s^{j-2}\right) - \left(\sum_{j=1}^{p} j\beta_j s^{j-1}\right) * \left(\sum_{i=2}^{p} (i^2 - i)\alpha_i s^{i-2}\right)\right|}{\left[\sqrt{\left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right) * \left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right) + \left(\sum_{i=1}^{p} i\beta_i s^{i-1}\right) * \left(\sum_{i=1}^{p} i\beta_i s^{i-1}\right)}\right]^3} \qquad (3\text{-}22)$$

where $\dot{r}(s,\alpha,\beta,p)$, and $\ddot{r}(s,\alpha,\beta,p)$ are themselves defined as vector valued functions with the following form,

$$\begin{aligned} \dot{r}(s,\alpha,\beta,p) &= \left\langle \left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right), \left(\sum_{j=1}^{p} j\beta_j s^{j-1}\right) \right\rangle, \\ \ddot{r}(s,\alpha,\beta,p) &= \left\langle \left(\sum_{i=2}^{p} (i^2 - i)\alpha_i s^{i-2}\right), \left(\sum_{j=2}^{p} (j^2 - j)\beta_j s^{j-2}\right) \right\rangle \end{aligned} \qquad (3\text{-}23)$$

Squaring the curvature constraint, yields a cleaner representation,

$$\frac{\left[\left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right)*\left(\sum_{j=2}^{p}(j^2-j)\beta_j s^{j-2}\right)-\left(\sum_{j=1}^{p} j\beta_j s^{j-1}\right)*\left(\sum_{i=2}^{p}(i^2-i)\alpha_i s^{i-2}\right)\right]^2}{\left[\left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right)*\left(\sum_{i=1}^{p} i\alpha_i s^{i-1}\right)+\left(\sum_{i=1}^{p} i\beta_i s^{i-1}\right)*\left(\sum_{i=1}^{p} i\beta_i s^{i-1}\right)\right]^3} \leq \left(\frac{1}{r}\right)^2 \forall s \in S(1)...S(n) \qquad (3\text{-}24)$$

The representation in (3-24) motivates *n* inequality constraints.

- **Proposition**: At a fixed value of $p, \bar{p}$, *and* $s, \bar{s}$, $K(\bar{s}, \alpha, \beta, \bar{p})$ is a non-convex function. Moreover, the feasible region due to a curvature constraint is also not a convex set.

- **Proof**:

  For $K(\bar{s}, \alpha, \beta, \bar{p})$ to be convex the following inequality must hold,

$$K(\lambda x+(1-\lambda)y) \leq \lambda K(x)+(1-\lambda)K(y),$$
$$\forall x, y \in \{\bar{s}, \Re^p, \Re^p, \bar{p}\} \; \forall \lambda \in [0,1] \qquad (3\text{-}25)$$

  The following counter-example proves the non-convexity. Setting $\bar{s}=2$, and $\bar{p}=1$, and the following values for the coefficients,

$$x=\begin{bmatrix}1\\0\\0\\2\\-10\\2\end{bmatrix}=\begin{bmatrix}\bar{s}\\\alpha_1\\\alpha_2\\\beta_2\\\beta_2\\\bar{p}\end{bmatrix}, \quad y=\begin{bmatrix}1\\2\\-20\\0\\0\\2\end{bmatrix}=\begin{bmatrix}\bar{s}\\\alpha_1\\\alpha_2\\\beta_1\\\beta_2\\\bar{p}\end{bmatrix} \qquad (3\text{-}26)$$

  Now, choose $\lambda=0.5$, implying $K(.5x+.5y)>0$ and $\lambda K(x)+(1-\lambda)K(y)=0$.

- **Result**: Each constraint in the NLP is defined at a fixed value of *p*, and *s*. $K(\bar{s}, \alpha, \beta, \bar{p})$ is not convex for a fixed choice of *p*, and *s*. Therefore, the level sets of a non-convex function are also not convex sets. Therefore, the NLP is non-convex.

The NLP is not a convex program and thus cannot be assured of finding globally optimal solutions. Nevertheless, experience in solving the constrained optimization model without the curvature constraints suggests that curvature constraints rarely this problem due to the data used in this UAV application. Therefore, if the heading of the vehicle is the same as the heading of the jagged path at *s=0*, then, the curvature constraint will likely be less of a problem. The hybrid approach in Section 3.5 establishes this condition. By establishing these conditions, the optimization model can be solved without the non-convex curvature constraints. If any of these constraints are binding then the non-linear programming code can use the solution from the optimization problem without the curvature constraints, as a starting point.

Other approaches may attempt to discretize and estimate the curvature constraint. This discrete representation may then be easier to handle within the optimization framework. Curvature is defined as the ratio of the tangent vector's rate of change to the rate of change in arc length. If a discrete representation is chosen, then the change in arc length between two points can be approximated as the distance between these points. Furthermore, the change in the tangent vector can be approximated as the difference in the tangent vector from one point to another. Since, the location of a point is dependent on $\alpha$, *and* $\beta$, the discretized version is still complex. A further estimate of arc length exists in the data vector, ***S***. Essentially, ***S*** is the arc length of the jagged path. If the mapping from the jagged path to the smooth trajectory through the polynomial functions maintains ***S*** as an estimate of the arc length of the vector valued function, then the curvature constraint can be approximated. The curvature constraint can be thought of as,

$$\frac{\left\|\dot{T}_i\right\|}{\dot{s}_i} \le \frac{1}{r} \forall i \in \{1 \ldots n\} \tag{3-27}$$

where, $\dot{s}_i$ is the rate of change in the arc length at point $i$, and $\dot{T}_i$ is the rate of change in the tangent vector at point $i$.

The tangent vector is essentially the first derivative, so its derivative is essentially the second derivative. This implies that a second difference approximation can be used for $\dot{T}_i$.

$$\left\|\dot{T}_i\right\| = \left\langle \sqrt{\left[f(s_{i+1}) - 2f(s_i) + f(s_{i-1})\right]^2 + \left[g(s_{i+1}) - 2g(s_i) + g(s_{i-1})\right]^2} \right\rangle$$
$$\dot{s}_i = \frac{S_{i+1} - S_{i-1}}{2} \tag{3-28}$$

Let $\alpha$ represent the row vector of polynomial coefficients in *f*, and $\beta$ represent the row vector of polynomial coefficients in *g*. In addition, define

$$\Delta S_i = \begin{bmatrix} S_{i+1}^0 \\ S_{i+1}^1 \\ \vdots \\ S_{i+1}^p \end{bmatrix} - 2 * \begin{bmatrix} S_i^0 \\ S_i^1 \\ \vdots \\ S_i^p \end{bmatrix} + \begin{bmatrix} S_{i-1}^0 \\ S_{i-1}^1 \\ \vdots \\ S_{i-1}^p \end{bmatrix}$$
$$d_i = \left(\frac{\dot{s}_i}{r}\right)^2. \tag{3-29}$$

Some simple arithmetic yields the following approximation of the curvature constraint.

$$\left(\alpha(\Delta s_i)\right)^2 + \left(\beta(\Delta s_i)\right)^2 \le d_i \tag{3-30}$$

The proof for convexity of (3-30) is straightforward. Specifically, both terms inside the squares are linear functions. Squaring these linear functions maintains the convexity of the resulting terms. Next, adding a positive multiple of a convex function to a positive multiple of another convex function maintains the resulting function's convexity. Finally, the level set of any convex function is a convex set.

The non-linear program is a purely convex program. However, the full non-convex NLP is not equivalent to this convex program. The convex program will find an optimal answer, but not the optimal answer for the non-convex problem. The reason for the disparity lies in the discrete approximation of the curvature constraints. In addition, the convex curvature constraints cannot be evaluated at the start and end point due to the definitions in (3-28). However, if enough points are chosen this limitation will probably be of little importance. The greater the number of points the more accurate the estimate of curvature will be. However, every new point will add another constraint to the model. This convex NLP may be difficult to solve due to the large number of constraints.

## 3.5 Hybrid Approach

In this section, modifications to the smoothing approach are made. The resulting hybrid approach can be used as a pre-processing step for either the unconstrained or the constrained optimization approach. The premise of this approach is that the heading constraint forces the curvature constraint to be binding. Clearly, if the desired heading is different than the heading of the jagged path, the constrained optimization will attempt to force the smooth trajectory to turn in the direction of the jagged path. If the difference is large, intuition provides that stress is placed on the curvature of the path near the initial point. The hybrid approach eliminates the difference in the desired heading and actual heading before the optimization is performed. Thereby, the stress on the curvature constraints is reduced in this region.

### 3.5.1 Introduction

The advantages to the least-squares method, and the constrained optimization method without the curvature constraints are many. The most important advantage to the least-squares method is computational speed. For the constrained optimization, the advantage relies in the ability to more accurately model the problem. The unconstrained optimization, can accommodate the START-END constraint, with careful adjustments to the weighting vector. The constrained optimization approach is able to handle both the START-END and START-HEADING constraints fully. However, the least-squares technique is unable to handle the START-HEADING or the CURVATURE constraint. This section details a method of forcing the START-HEADING constraint for either the unconstrained or constrained optimization in a much different way than seen in Section 3.4. Similarly, to before, the CURVATURE constraint will hopefully be nonbinding for the resulting solution using either the constrained (w/o curvature constraints) or unconstrained optimization. If not, it needs to be handled as outlined in Section 3.4.2

### 3.5.2 Methodology

The focus of this method is dealing with the START-HEADING constraint. The objective of any smoothing technique is to stay as close as possible to the path, *p*. In light of the objective, if the initial heading must equal some predefined heading, the vehicle would benefit by turning back towards *p,* as quick or with as little deviation as possible. If the smoothing of the path was only performed for a few of the initial points along the jagged path, then the turn back towards *p* could be accomplished with the minimum turn radius possible. Essentially, if the objective function is only defined for those points along the turn, the optimal solution would be to turn back to the path with as little deviation as possible. However, upon rejoining the path the heading of the path must be taken into account. Thereby, two turns must occur. The first turn takes the vehicle back towards the path; the second turn is in the opposite direction and ensures that the heading of the vehicle is correct upon the rejoin. The second turn also occurs with the minimum turn radius. Figure 3.5 is a graphical representation of the two turns. It should be noted that the solution, as illustrated, is the best the constrained model (over all-possible functions, f and g) could perform in this region of the path.

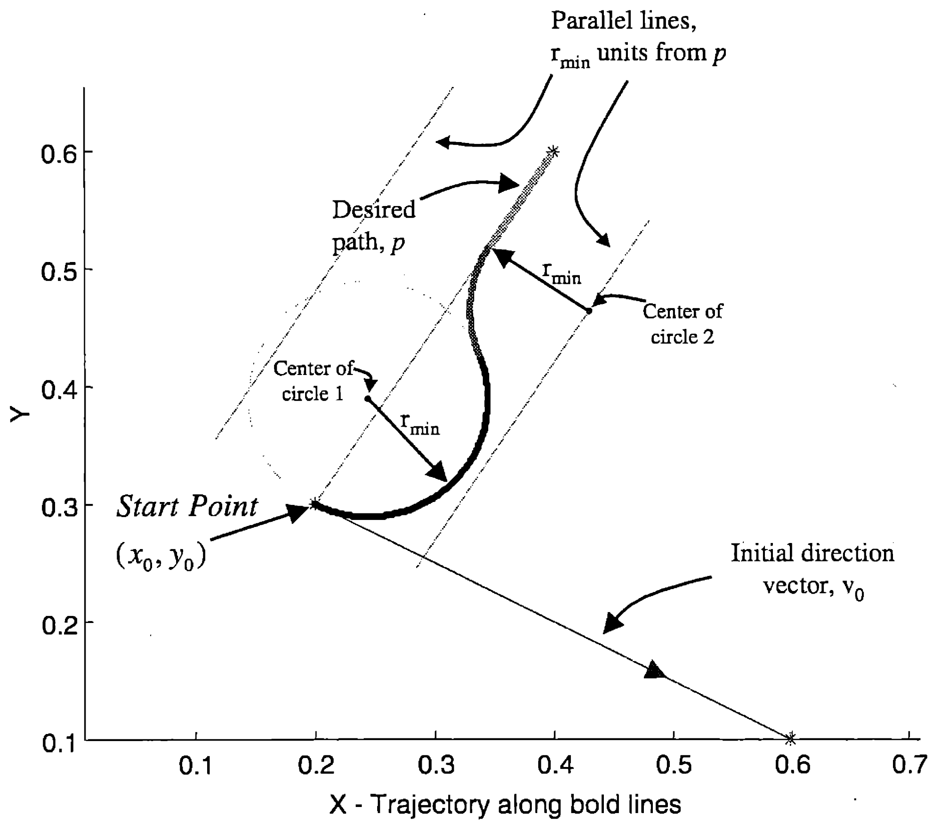


**Figure 3.5 Illustration of 2-Turn Procedure**

The solution to the 2-turn problem boils down to finding the centers of the two circles. One major assumption is made to simplify the solution; the path, *p*, although previously defined as being jagged is now defined to be a straight line emanating from the start point. As long as the original path is straight within *3r* (worst case) units from the start point, this assumption holds. However, the method could be adapted to the case where *p* is jagged within this bound. Figure 3.5 helps describe the following algorithm:

- Define two lines parallel to *p* that are *r* units in distance from *p*.
- Find the center of circle 1. (A result of the initial direction and start point.) Two answers will result. Choose the center whose distance to some point along *p* (not the start point) is minimum.

- Find the points on the parallel lines *2r* units from the center of circle 1. One of these 4 points is the center of circle 2.

- Choose the center of circle 2 such that the heading of the vehicle at the rejoin point is aligned with the heading along *p*.

If the original path is not a straight line from the start point to the rejoin point, a modification to the 2-turn algorithm could alleviate the problem. Essentially, step 1 of the algorithm would need to create a set of line segments that are parallel to the jagged path. The remaining steps still apply. In the particular application chosen, the minimum turn radius is small enough that the *3r* assumption is not violated.

The following question remains – *How does one smooth the path after the 2-turn solution?* The first step is to locate the rejoin point and then replace the start point in ***P*** with this rejoin point. Assuming the original path is a straight line from the start point to the rejoin point, all that is needed is to replace the start point with the rejoin point. From this point, all the developments of both the constrained and unconstrained methods apply. For the unconstrained method, two careful adjustments must be made in order to ensure that the heading of the vehicle at the rejoin point is along *p* and the rejoin point is the starting point for the smooth trajectory. This adjustment is made by carefully adjusting the weighting vector. Instead of weighting just the first point (to ensure the start point is correct), the first *k* points need to be weighted. The choice of *k* is arbitrary (however, *k* must be less than the number of data points between the vertices of *p*.). In the context of this application, the first four data points were weighted in a decreasing fashion. For instance, letting *w(1)=10,000* as previously described, *w(2)=5000*, *w(3) = 1000*, and *w(4)=100*, ensured the initial heading from the rejoin point matched the heading of *p* at the rejoin point.

Using the constrained model, the specification of the start point is now the rejoin point. The START-HEADING constraint is still needed, to create a smooth path. However, now the vehicle is heading in the direction of the jagged path. As previously mentioned, this condition places less burden on the curvature of the path near the start point.

### 3.5.3 Possible Modifications to 2-Turn Procedure

The 2-turn procedure gives a method for smoothing a path around a jagged section of the graph. It should be noted that similar methods could be used to smooth the path, eliminating the need for the entire optimization approach to the path-smoothing problem. Modifications to the 2-turn procedure could be made so those jagged portions of the entire path are smoothed. The methods as developed, yield a solution whose worst case error is twice the turning radius of the vehicle. Similar to the optimization approaches, these methods will begin to experience larger deviations from the path when the jaggedness of the path increases. The larger the heading differences from linear segment to linear segment, as well as the number of linear segments comprising the path is a measure of the jaggedness. Increasing either both or one of these parameters will begin to stress a 2-turn type procedure (and even the optimization type methods).

In this thesis, extending the 2-turn procedure to the entire path is not studied. However, future research should compare the performance of a 2-turn type procedure to the performance of the optimization methods as developed in this chapter.

## 3.6 Conclusions

This chapter provided an overview of how to formulate the path-smoothing problem. Other methods may exist which solve this problem in a different manner. Specifically, one could eliminate the entire optimization of the path and rely on the vehicle's controller to maintain a specific ground track (even if that ground track is infeasible). The techniques developed in this chapter attempt to create a path, which is feasible for the vehicle. The vehicle's controller is still needed to maintain this path, but hopefully the real-time control algorithms better handle the smooth trajectory than an infeasible jagged trajectory. Another approach could modify the 2-turn procedure in order to eliminate any jaggedness of the path. Even though other methods may exist, the optimization approach has been created as a possibility for the path-smoothing problem.

The vehicle simulation is written using MATLAB, and the hybrid approach using the constrained optimization as part of that simulation. MATLAB has a built-in optimization function that is capable of handling the constrained optimization (without the curvature constraints) as presented. The implementation ignores the curvature constraint, as it is rarely binding in practice. The fact that the constraints are not binding is a function of the data inherent in the jagged paths. If the paths are sufficiently jagged, then clearly the curvature constraint would become more of an issue. If these techniques are applied in other situations then the appropriate handling of these constraints is needed. The curvature constraint can be included in either its non-convex form, included in its estimated convex form, or eliminated. If the constraint is eliminated, then the vehicle's controller needs to be able to correctly compensate for any situations where the path violates the minimum turn radius of the vehicle. In the simulation discussed in this thesis (Chapters 4 and 5), if the constraint is binding then we need to assume that the controller is able to compensate.

The convex NLP as formulated should provide an optimal answer after a solution is found. However, computational results were far from satisfying for this model. The results for both the constrained model without the curvature constraint, and the convex formulation with the curvature constraints are briefly presented in Chapter 4. The results suggested that the convex NLP with the curvature constraints while seemingly well behaved in formulation is not well behaved upon actual implementation. Therefore, the MATLAB routine designed for the constrained model without the curvature constraints was used.

# Chapter 4

# Path Planner

This chapter provides an efficient approach for a UAV, with the task of tracking a moving target. Elements from the previous two chapters are integrated and developed further in order to produce the path planner. In addition, the path planner is developed in the context of a specific application. The application chosen is an underwater autonomous vehicle whose environment of operation is Narragansett Bay, Rhode Island. However, the approach is not vehicle specific and could be generalized to any vehicle.

The framework for the moving target path planner is not specific to a particular environment or vehicle. However, applying the framework to a specific application shows the power of the moving target path planner. In order to demonstrate the moving target path planner, the concepts, as presented in this chapter, are implemented in a simulation. The results from the simulation are included in Chapter 5.

A broad sense of the algorithm can be gained by imagining the target dropping breadcrumbs as it travels through its environment. The path planner, as developed, instructs the seeker to then follow that set of breadcrumbs. However, it does not necessarily follow the same path taken by the target. The path taken by the target may be infeasible for the seeker. The path may be infeasible for the seeker due to the target having more knowledge of the environment. With greater knowledge, certain obstacles that are present for the seeker are not obstacles for the target. Furthermore, the target may be capable of performing maneuvers that the seeker is not. Thus, the seeker cannot strictly follow the path of the target. Therefore, the seeker will travel between breadcrumbs in a least cost fashion (i.e. as defined by cost metric).

## 4.1 Definitions

1. Network – The network is a 2-D grid map. The neighborhood size is 16, and is defined the same as in Chapter 2.

2. Cost – The cost metric chosen is distance. The cost along any arc is the distance from the center of the two grid cells connected by the arc.

3. Current – The current (or wind for unmanned aerial vehicles) is defined as a fixed value for a specific grid cell, at any instant in time.

4. Terminal State – When the seeker vehicle has no path information to follow, the terminal state exists. In the context of the path planner, the state will always occur when the seeker reaches the current goal. New path information may not be available for one of two reasons. 1)The seeker vehicle failed to acquire a target position update. 2) A new target position has been acquired, but the path planner is busy computing the new path.

5. Errors - a) Position error estimate is defined as the error in estimating the current position. In many UAVs an inertial navigation system is used to maintain an estimate of the current position. b) Path smoothing error is defined as the deviation in the smooth path from the jagged shortest path. c) Navigation error is defined as the error associated with the vehicle failing to exactly follow a commanded track. The vehicle may be commanded to travel along a straight line track, but due to unforeseen circumstances the vehicle deviates from this desired track.

## 4.2 Assumptions

1. Map – The map is known a-priori, and is fixed for the life of a mission.

2. Speeds – The magnitude of the seeker's velocity, $\|V_S\|$, is assumed constant as is magnitude of the target velocity, $\|V_T\|$.

3. Current – The magnitude of the current is less than $\|V_S\|$. Otherwise, the seeker vehicle may not be able to make forward progress for a desired ground track.

4. Obstacles and the Map – The obstacles in the map are assumed buffered by an amount that will maintain the safety of the vehicle. The obstacles must be buffered in order to account for the following errors: errors in position estimate, errors in path smoothing, and navigation error.

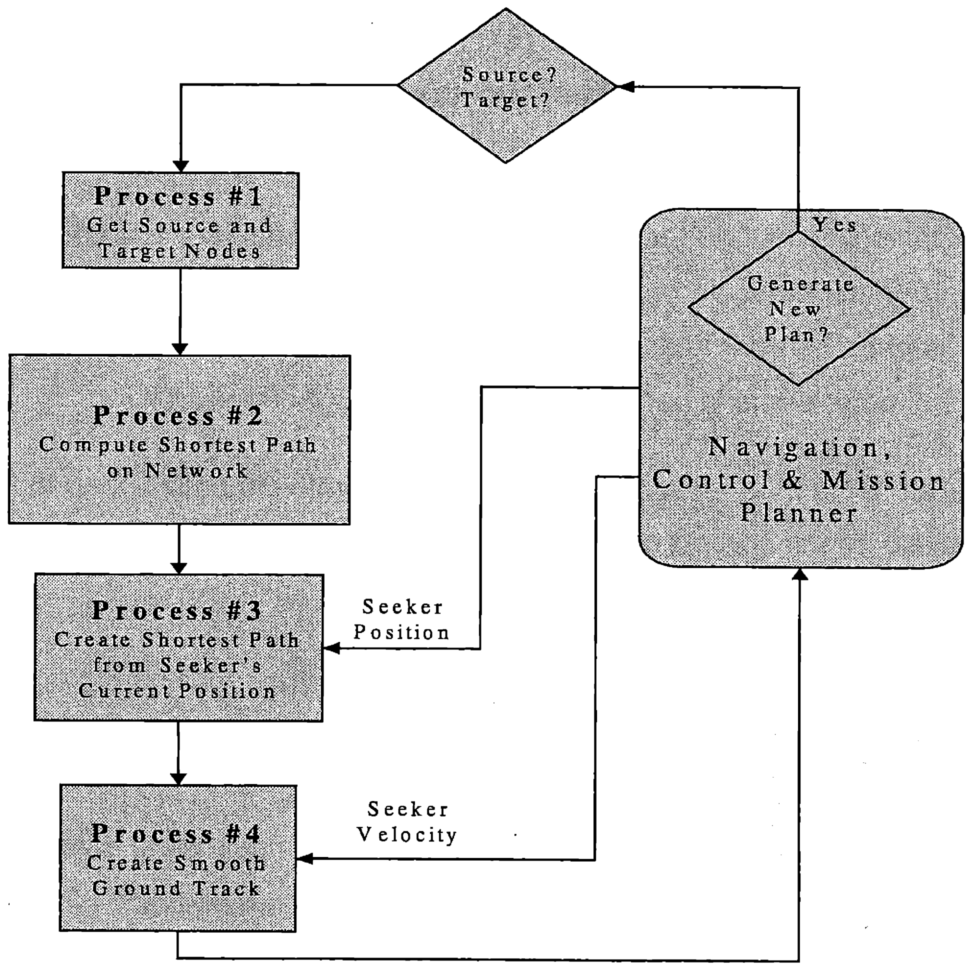
5. Path Existence – If a path is desired from node i to node j, then the path must exist. In the context of the path planner, a path will never be desired that cannot be found.

6. Terminal State – A vehicle behavior exists such that either of the two following conditions holds. 1) The seeker vehicle is able to progressively slow down such that once the goal of the current path is reached a new path to a new goal is assured of existing. Or, 2) if the terminal state does exists, while the vehicle awaits a new plan, the vehicle is assumed to travels to nodes for which the backpointer of that node is defined. To motivate the second condition, suppose the terminal state occurs, and the vehicle travels to a node that does not have a defined backpointer. Then, a path cannot be created from that node.

7. Initial State – The initial velocity of the vehicle is assumed to be zero. In addition, the initial heading of the vehicle is assumed to lie in the direction of the first shortest path.

## 4.3 Algorithm Description

The following flow chart gives a high level description of the path planner. The details of how the path planner operates in the moving target scenario are a function of how the source and target are chosen, and how the path is created (process #3).



Source?
Target?
Process #1
Get Source and Target Nodes
Process #2
Compute Shortest Path on Network
Process #3
Create Shortest Path from Seeker's Current Position
Process #4
Create Smooth Ground Track
Seeker Position
Seeker Velocity
Yes
Generate New Plan?
Navigation, Control & Mission Planner

**Figure 4.1 Description of Path Planner**

In general, the path planner as illustrated in Figure 4.1 is capable of solving the shortest path problem for the moving target scenario. As shown, the path planner is a sub-function of the UAV. If the mission planner is in moving target search mode, the four processes are executed repeatedly through time.

An example of how the above flowchart works in practice is helpful in understanding the way a moving target planner works. First, the mission planner decides that a moving target needs to be tracked. At this point, the mission planner invokes the moving target functionality of the path planner. Process #1 stores the most recent target location as the target, and the current position of the seeker (itself) as the source. Process #2 then computes the shortest path from the target to source. In addition, process #2 stores the backpointer list for later iterations. In general, Process

#2 takes a non-zero amount of time, so process #3 must take the new position of the seeker and compute the shortest path from this new location. However, at the first iteration of the algorithm the seeker is assumed to have zero velocity. Therefore, the current position of the seeker is the source in the shortest path problem. Process #3 uses information contained in the backpointer list to aid in finding the actual shortest path. Now Process #4 takes the path and smoothes it. Process #4 does nothing to the backpointer list and is passed back to the mission planner. The smooth ground track is also given to the mission planner, which then passes it to the navigation and control algorithms. These algorithms execute the smooth ground track, until the mission planner decides a new target location needs to be integrated.

The mission planner invokes the path planner again. At this point in Process #1, the source of the new path is not the current vehicle position. Rather, the source is chosen as the old target location, and the target is chosen as the new target location. Process #2, in computing the shortest path, uses the old backpointer list. Changes are made to this list if the shortest path computations warrant the change. The remaining processes are carried out in the same manner as in the first iteration.

The following sections detail the processes as illustrated in Figure 4.1. The first three processes are grouped together in section 4.4 on shortest paths. The final process is described in section 4.5 on trajectory smoothing.

## 4.4 Shortest Path

At the network level, the moving target search is solved as a series of static shortest paths. In order to solve the moving target problem, one could implement successive shortest path algorithms, where the new optimal path is computed from the current position of the vehicle to the target's location. As previously illustrated, the number of nodes searched in a grid network increases as the distance from source to goal increases. If the target and seeker are far apart, then computing successive shortest paths from seeker to target may be inefficient. In addition, implementing the shortest path, planned in this manner, will not be feasible due to the computation time of the network shortest path algorithm. Specifically, after the shortest path algorithm completes, a certain amount of time has expired. Assuming the vehicle is travelling at some non-zero velocity during the path computation, the current position of the vehicle is no longer the source as provided to the shortest path algorithm. Therefore, constructing a shortest path from the actual position of the vehicle to the target (after the path computation) may not be feasible.

Instead of computing paths from the seeker position to the target position, the planner takes a different approach. As outlined in the description of the flowchart, Figure 4.1, only the first path is computed from seeker to target. All other paths are found by searching from the previous target location to the current target location. Then the path is found from the seeker position by following the set of backpointers to the most current target position. If the backpointer list provides no improvements over the entire length of the mission, then the resulting trajectory will be the shortest path from the initial seeker position, and the connection of all shortest paths planned from a previous target location to the next target location.

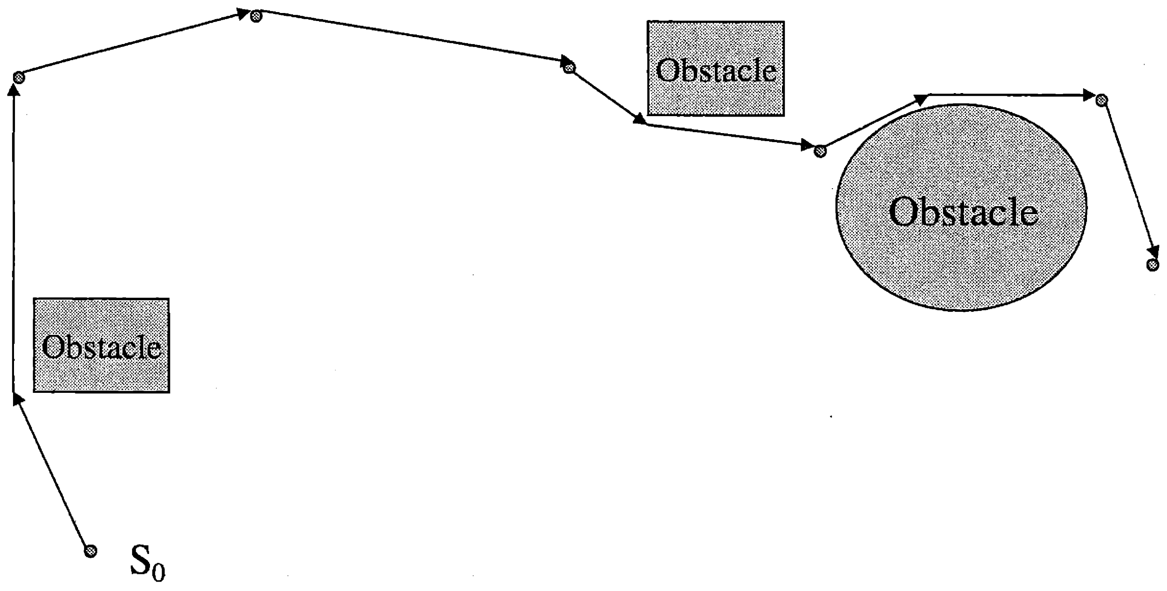


**Figure 4.2 Resulting Trajectory with No Information in Backpointer List**

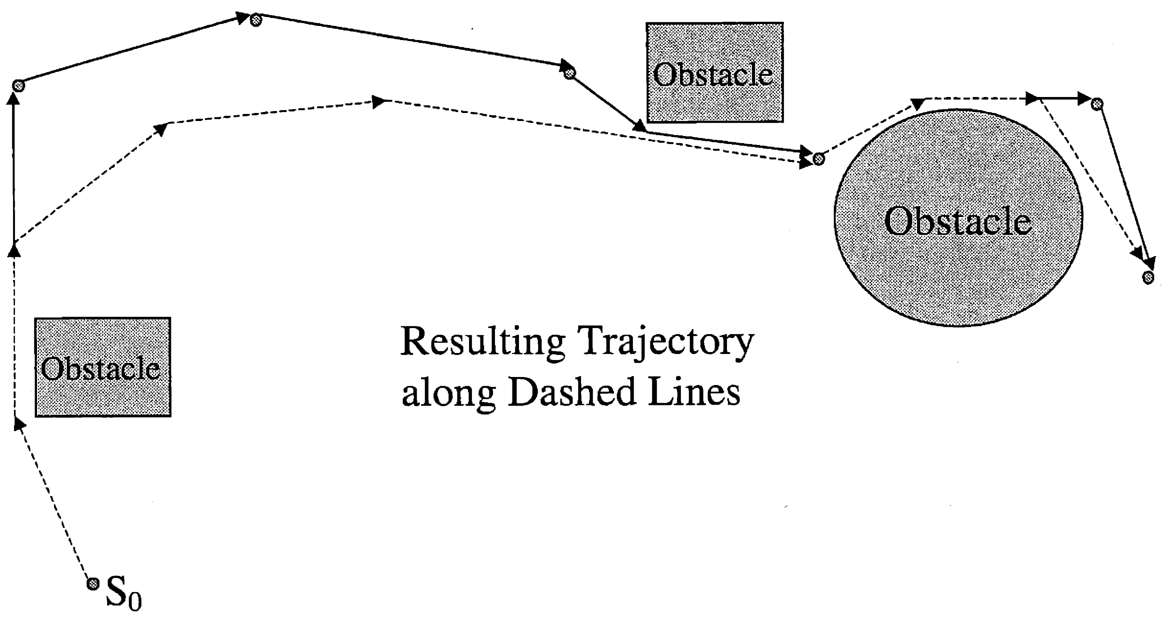


**Figure 4.3 Resulting Trajectory with Improvements Due to Backpointer List**

In Figure 4.2 and Figure 4.3 the initial seeker position is labeled $S_0$ and all other points are the target locations. Figure 4.3 shows the benefit of updating the path from the seeker to the current target location using the backpointer information.

### 4.4.1 Description

The following algorithm addresses both the inefficiency of computing full plans form source to goal, and the problem of how to construct the path in light of a constantly moving seeker.

```
algorithm moving target shortest path;
begin
1.  initialize $\Delta t$;
2.  backpointer(j)=0 $\forall$ j $\in$ N (N = arcs in G)
3.  t:=0;
4.  seeker_speed=0;
5.  for i = 1 to $\infty$
6.      path(i)=null;
7.      target(i) = target position at time t;  (Sensor update)
8.      if t=0 then source = initial seeker position;  else source(i) = target(i-1);
9.      time_to_compute=curent_time;
10.     find shortest path(i) from target(i) to source(i)
11.         while computing shortest path
12.             if t>0 follow path(i-1) until shortest path completes; else wait for shortest path algorithm;
13.         end
14.         update backpointer;
15.     end;
16.     time_to_compute=current_time-time_to_compute;
17.     get current_position;  (Navigation update)
18.     compute path(i) by following backpointers from current_position to target(i);
19.     seeker_speed=seeker_operational_speed;
20.     while t < t + $\Delta t$
21.         follow path(i);
22.         t = current_time;
23.     end;
end
```

**Figure 4.4 Moving Target Algorithm**

The above algorithm may seem complicated, but it is a simple method of computing a continuous path from the moving seeker to the moving target. In order to illustrate the algorithm, a few iterations of the algorithm are discussed. In the following description the number on the left refers to the line number in the moving target algorithm.

- INITIALIZATION

  1. In order to initialize the algorithm, the sensor update time, $\Delta t$, is set
  2. The backpointer for every node in the graph is initially set to zero,
  3. The current time is set to zero.
  4. Assuming the seeker is stationary upon beginning the algorithm the seeker speed is set to zero.

- LOOP

6. The current path is set to null.

7. The current location of the target is acquired from the mission planner via a sensor update.

8. The source for the shortest path algorithm is chosen. At the first iteration it is the initial position of the seeker. At later iterations the source is the goal of the previous iteration (i.e. the target's location at the previous iteration)

10. Using the information from step 8, the shortest path algorithm begins to look for the shortest path from goal to source.

12. While performing step 10, the vehicle continues on the previous path. If this is the first iteration of the algorithm, the seeker waits.

14. Once step 10 completes, an updated backpointer list should be available.

17. The path planner assesses navigation for the best estimate of the seeker's current position.

18. A path is found from the current position of the seeker, to the target, using the information contained in the backpointer list.

19. The speed of the seeker is set to the speed as determined by the mission planner.

20 –23. The seeker follows the new path until a target update occurs, in which case the path planner algorithm loops beginning with step 6.

The entire path (over all iterations of the path planning algorithm) followed by the seeker, in the worst case, will be composed of the shortest path from the seeker's initial location to the target's initial location, and all the shortest paths computed from a target location to the following target location. The path will likely be better (i.e. lower cost) than the worst case path because information in the backpointer list may provide improvements over the worst case path. Examples of this improvement are seen throughout the next chapter, which discusses the performance of the path planner

In the algorithm, $\Delta t$ must be chosen carefully. Specifically, $\Delta t$ must be greater than time it takes to compute the path. The upper bound could be as large as desired. However, the larger the value of $\Delta t$ the worse the algorithm performs. In terms of a UAV, $\Delta t$ refers to how often new target information is incorporated into the moving target scenario. The sensors onboard a UAV may acquire target information at a rate faster than the computation time of the shortest path algorithm. For instance, the sensors may acquire target location every 3 seconds. However, the shortest path algorithm takes 5 seconds to complete. This condition is not a problem in implementation because the target location does not need to be incorporated at the rate it is updated. Even though the update to the target location occurs every 3 seconds, one can choose $\Delta t > 5$ seconds, and use the best information about the target's location available. Clearly, the seeker always will travel to the target's last known position. In addition, due to the nature of the computed paths, the behavior of the seeker's trajectory is likely to be close to the target's trajectory. If the target is travelling according to a similar cost metric as is the seeker, these trajectories will tend to be more similar. Therefore, the seeker tends to follow the target's trajectory. Because, the seeker is always computing paths to the target's last known position, and the manner in which the paths are computed, implies that the seeker will exhibit the desired trailing behavior. There is no reason that the target updates have to occur at fixed intervals. The mission planner or some higher level process on-board the UAV could specify when the target update should occur. The analysis in Section 4.6 is based on fixed time intervals between target location updates. In Section 5.4.3 a variant to the fixed time interval method is discussed.

### 4.4.2 Illustration of Shortest Path

This section provides an illustration of the environment as well as typical shortest paths in that environment. The following figure represents the environment of operation.

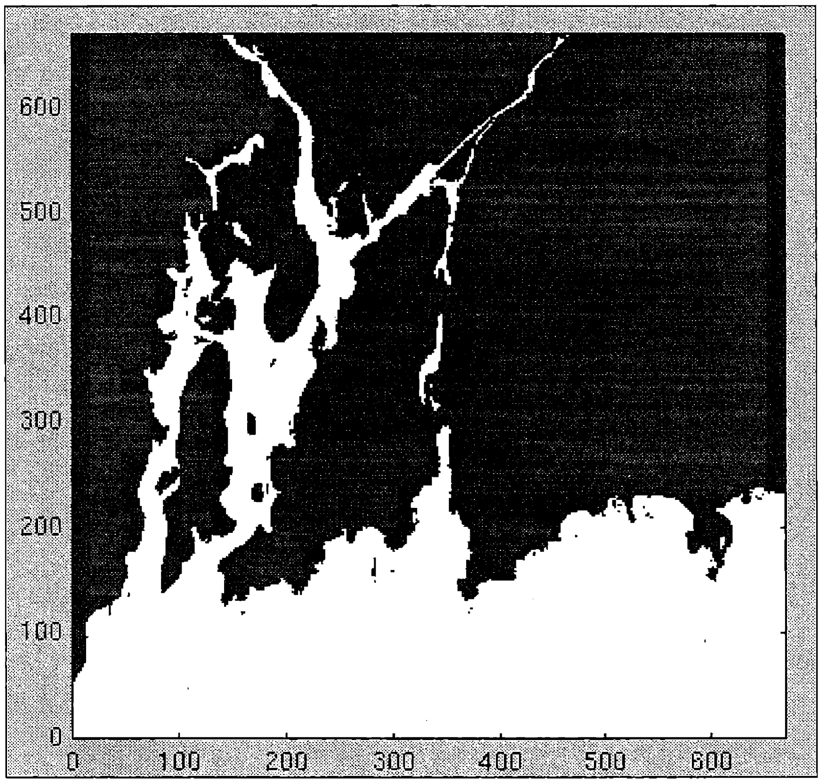


**Figure 4.5 Map of Narragansett Bay**

In Figure 4.5, the grid size is 50 meters by 50 meters. The map is 691 x 691 grid cells, resulting in 477,481 nodes in the graph. Furthermore, the number of free grid cells is 157,364, or in other terms approximately 33% of the map is traversable. This results in 2,517,824 arcs in the graph. The dark area represents obstacles, and the white region represents free cells. Furthermore, some of the dark area may actually represent water, but at depths too shallow for the vehicle. This 2-D representation was created from data that contained the depth of the bay floor at each position on the map [2]. An acceptable depth for travel was chosen, and any depth not meeting this criterion was labeled as an obstacle.

The next figure illustrates how a typical shortest path is displayed on the map.

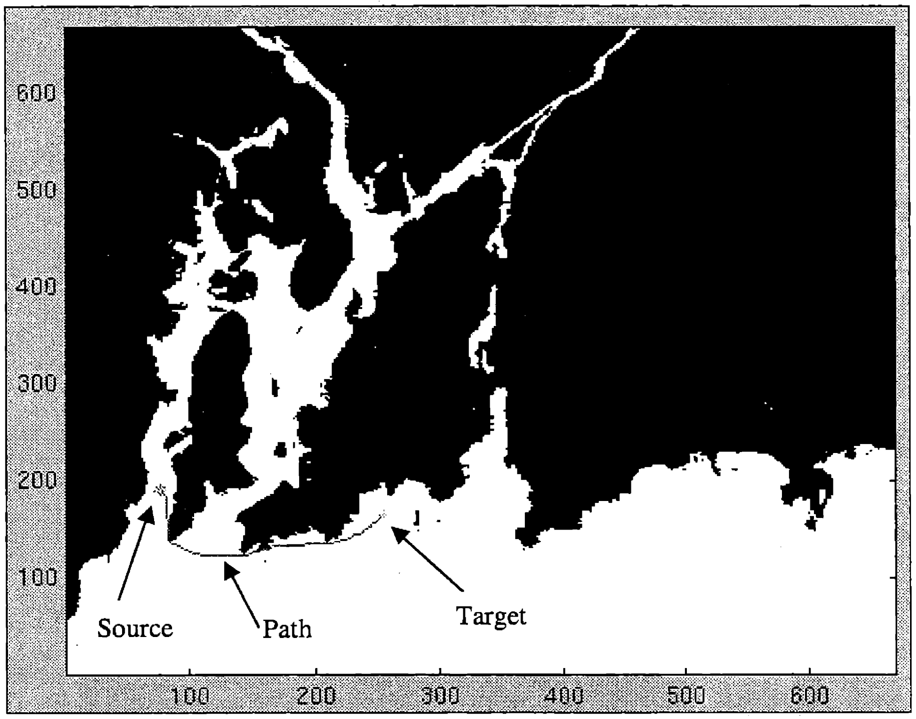


**Figure 4.6 Illustration of Typical Shortest Path**

### 4.4.3 Illustration of Backpointer Information

Once the shortest path algorithm terminates, more information than just the shortest path is available. If a node is labeled as permanent during the search a backpointer for that node exists. After the shortest path algorithm ends, this information may become very helpful. For example, in following a shortest path the UAV may depart that path for unforeseen circumstances but still remain in a grid/node that has been labeled as permanent. If the backpointer information is stored after the algorithm terminates, the UAV is able to follow a set of backpointers to the target. In the case of the algorithm as shown in Figure 4.4, the backpointer information provides information that will improve the path from the seeker to the moving target. Without this information, the seeker would travel to the target's first known location, and then follow a series of successive shortest paths from that location, to the following locations of the target. As the algorithm proceeds it maintains the backpointer list. For example, after the first plan, those nodes that are labeled permanent have a backpointer assigned to them. As a result of initialization, all other nodes have backpointers set to 0. During the second plan, the same backpointer list is used and is updated as necessary. Those nodes not permanently labeled, cause no change to the backpointer list. In reference to the grid maps and neighborhood structures as previously defined. The backpointer can be stored as a number from 1 to the neighborhood size. For example, when $|neighbor(x)| = 16$, the backpointer at any node is stored as a number from 1 to 16. The following figure is an illustration of the backpointer information after a single shortest path run. The different shades of gray represent the different values (1 to 16) at each grid cell. In addition, the darkest cells are assigned a value of 0. If the cell has a value of 0, then it is either an obstacle or it is not in the set of permanently labeled nodes (i.e. no backpointer associated with it).

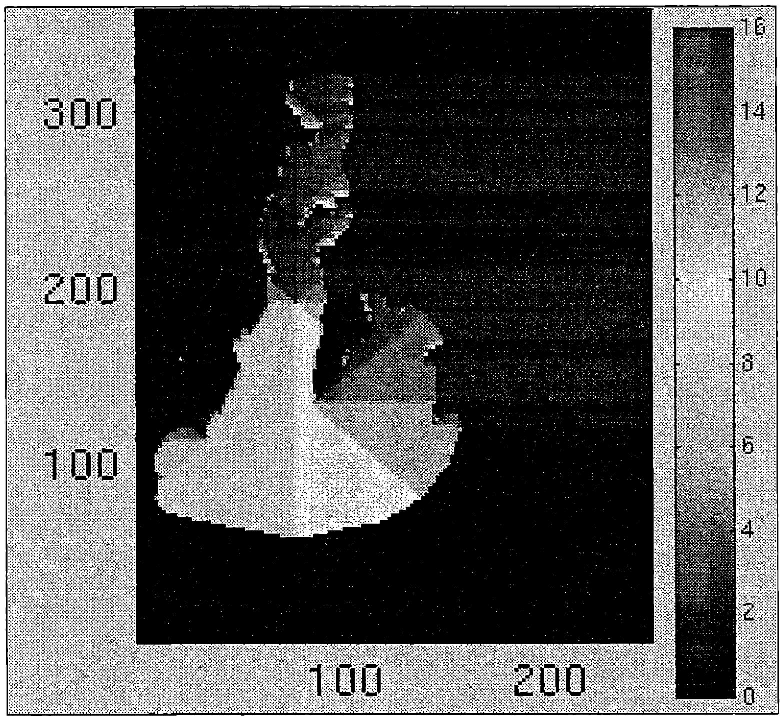


**Figure 4.7 Illustration of Backpointers**

The next figure is an illustration of the backpointer information after two shortest path calculations.

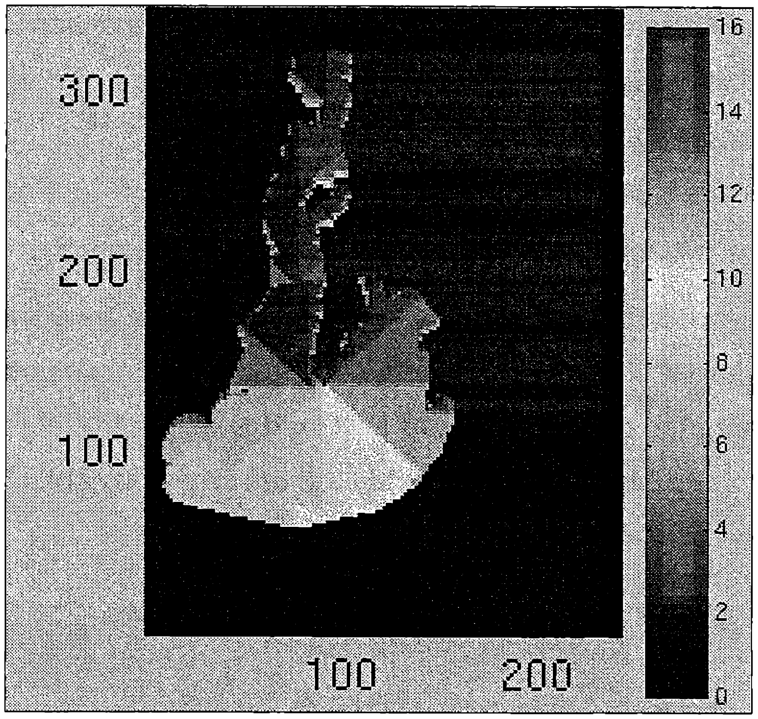


**Figure 4.8 Illustration of Backpointer after two Shortest Path Computations**

Both Figure 4.7, and Figure 4.8 have a legend, which explains which shade corresponds to the value of the backpointer at a grid location. The difference in the two figures corresponds to an area of the grid labeled with new backpointer information. This area is centered around the target (or goal) of the second shortest path computation.

It is easy to recognize that if the backpointers are maintained as described in Section 4.4.1, a path will exist from the seeker's current location to the most recently incorporated target location.

## 4.5 Trajectory Smoothing

This section provides a brief look at smoothed trajectories, and how two smooth trajectories are pieced together. The method used for smoothing is essentially the hybrid approach given in the chapter on path smoothing. The vehicle is assumed to be heading in the direction of the shortest path for the initial trajectory. The conditions for subsequent trajectories are established using the 2-turn procedure. Extremely important to any path smoothing procedures is the maximum error associated with the smoothing technique. As stated before this error is part of the error used to buffer the obstacles. The derivation of the maximum error associated with the 2-turn procedure is developed. The maximum error associated with the optimization procedure is computed from output of the optimization.

### 4.5.1 2-Turn Procedure, Description and Error Development

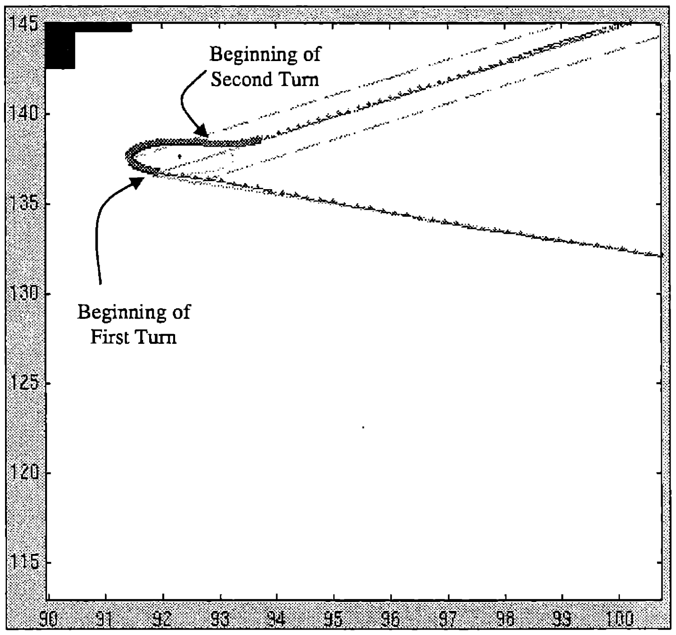


**Figure 4.9 Illustration of 2-Turn Procedure**

An important measure of performance for the 2-turn procedure is the maximum deviation from the desired path. The angle between the desired heading and the initial heading is the driving variable for the maximum error. The following picture shows diagrams the details involved in the 2-turn procedure.

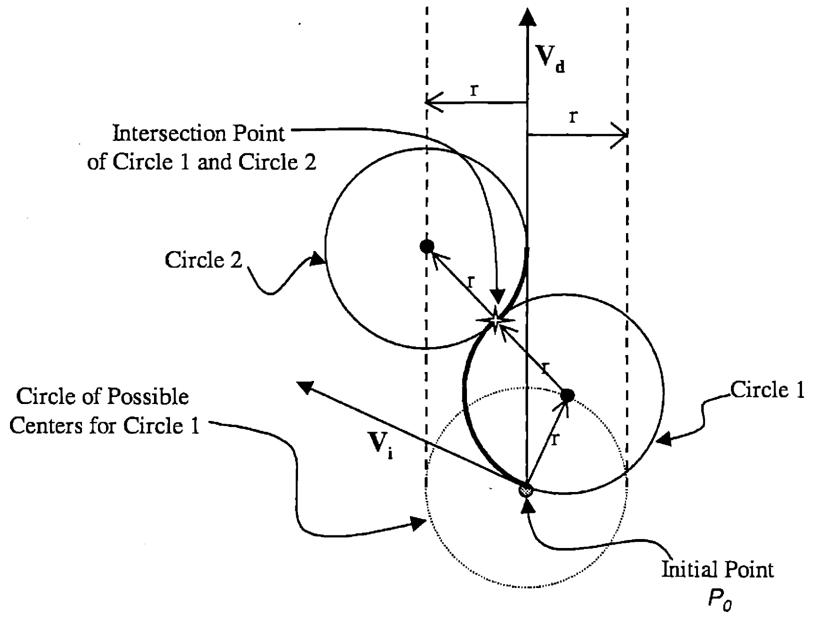

V_d
r
r
Intersection Point of Circle 1 and Circle 2
Circle 2
r
r
Circle of Possible Centers for Circle 1
V_i
r
Circle 1
Initial Point P_0

**Figure 4.10 2-Turn Diagram**

To review the procedure in terms of Figure 4.10 the problem is characterized by the following parameters.

1. Initial heading vector, $V_i$
2. Desired heading vector, $V_d$
3. Initial Point, $P_0$
4. Desired Path, composed of the ray defined by $P_0$ and $V_d$.
5. Turn Radius, $r$
6. Circle 1, and Circle 2
7. 2 Lines $r$ units from $V_d$

The following diagram is a magnification of Figure 4.10, and will be used to motivate the function that describes the maximum deviation from the desired path for the 2-turn procedure.

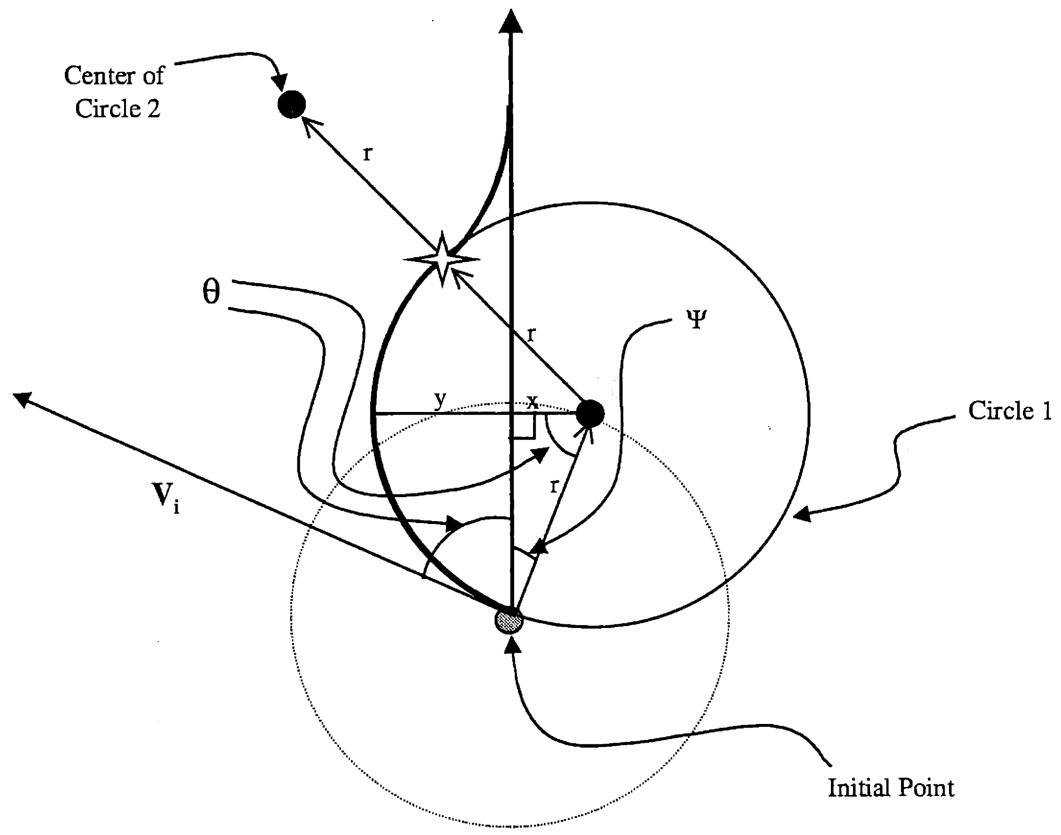


**Figure 4.11 Magnification of 2-Turn Procedure**

- **Proposition**: The maximum error, *y*, due to the 2-turn procedure is a function of the angle between $V_i$, and $V_d$, and the turn radius *r*.. This function is

$$y(r,\theta) = r\left(1-\cos(\theta)\right) \tag{4-1}$$

- **Proof**: The following proof draws heavily from Figure 4.11.

   1. We can always transform the coordinate system so that the desired heading vector, $V_d$ lies in the positive y-direction, the initial point is the origin, and the angle from $V_i$, to $V_d$ is preserved in magnitude and direction.

   2. The maximum deviation occurs on circle 1. When the turn is initiated along the second circle, the heading is convergent of the desired heading. All the subsequent headings along the turn on circle 2 are convergent on the desired path. Because of the convergent headings, the distance will strictly decrease throughout the entire turn on circle 2. Therefore, the maximum error on circle

2 is at the intersection point. Since the distance from the desired path is decreasing across the intersection point, the maximum distance occurs on circle 1.

3. The maximum deviation is *y*, as shown in the above figure.
4. By definition, $y + x = r$.
5. $\Psi = 90 - \theta$
6. By law of sines, the following equality must hold,

$$\frac{\sin(90-\theta)}{x} = \frac{\sin(90)}{r} \tag{4-2}$$

7. By trigonometric identities and manipulation of (4-2),

$$x = r\cos(\theta) \tag{4-3}$$

8. With the result in (4-3), and the definition, $y + x = r$,

$$y = r\left(1-\cos(\theta)\right) \tag{4-4}$$

- **Result:** We have shown that $y(r,\theta) = r\left(1-\cos(\theta)\right)$ as proposed. And the largest value *y* can attain is *2r*.

A numerical approach was performed to validate (4-1). The results are shown in the following plot.

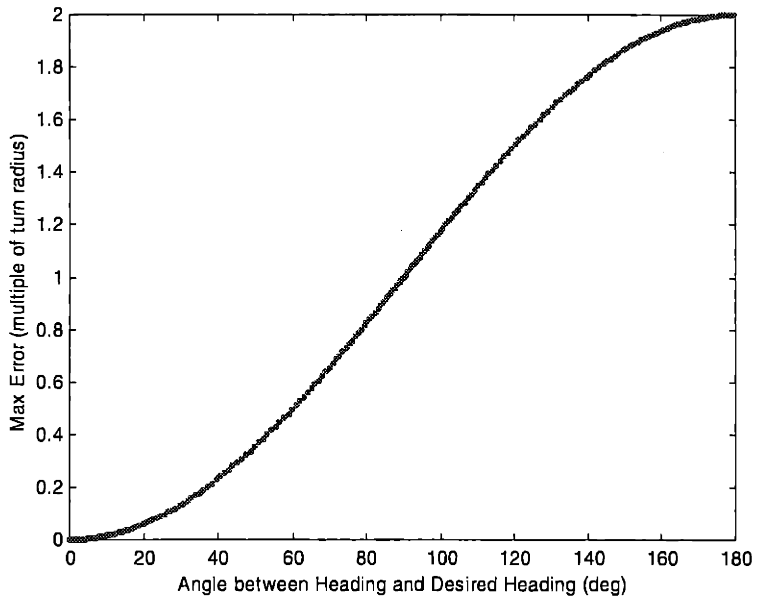


**Figure 4.12 Max Error Due to 2-Turn Procedure**

This plot is essentially the plot of (4-1) with $r = 1$, verifying the previous argument.

*Why is this error important*? This error is one of two errors that can be used to access whether a 2-D grid map is sufficient for path planning. If this error, is larger than the error due to smoothing a path then the obstacles only need to be buffered by *2r*. However, the *2r* error may never or rarely happen in a vehicle's mission. Therefore, the obstacles may only need to be buffered by some amount less than *2r*. Simulation results could provide a distribution, which characterizes this error. With this distribution, one could choose some deviation that encompasses a large percentage of the distribution's area. After finding this number, the obstacles could then be buffered by this amount. In the unlikely case that a *2r* error does arise, the vehicle could rely on higher level processes to maintain the vehicle's safety.

Once the 2-turn procedure is complete, the conditions are satisfied for performing the rest of the hybrid smoothing technique. The following picture shows the initial smoothed trajectory. The initial trajectory does not require the 2-turn approach because of the assumption concerning the initial heading vector. If this assumption is relaxed, then the 2-turn approach could be used for the initial trajectory.

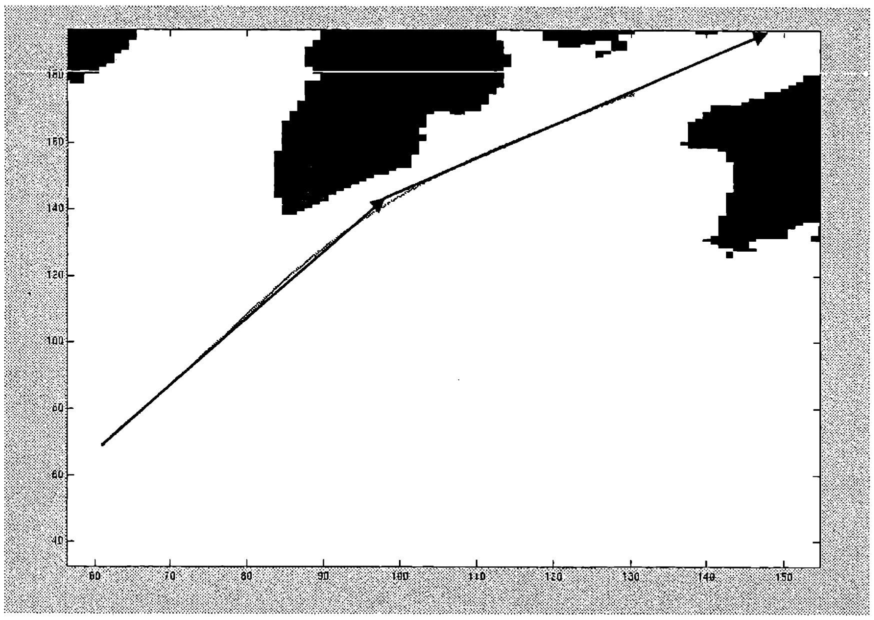


Figure 4.13 Illustration of Smooth Path

In Figure 4.13, the smoothed path and the shortest path are almost exactly the same path. The shortest path is composed of the two arrows, and the smooth path terminates at some point before the second arrow is reached. The next figure, Figure 4.14, is a magnification of the seeker's trajectory in order to illustrate the details of a typical trajectory.

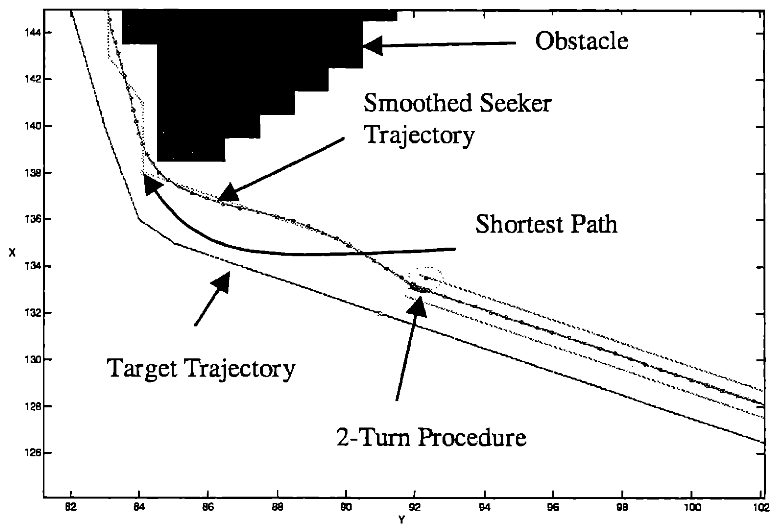


Figure 4.14 Illustration of a Continuous Smooth Trajectory

### 4.5.2 Constrained Optimization (Smoothing), Description and Error Development

The model chosen to smooth the function is the constrained optimization model (per Chapter 3) without the curvature constraints. The limitations of not including the curvature constraints are noted. However, the curvature of any trajectory is rarely violated. In the rare instance that the curvature of any smooth trajectory is violated, it is assumed that the vehicle controller is able to follow a ground track that is close to the desired ground track. As before, the model is formulated as, (see Section 3.4.2)

$$NLP: \quad \min_{\alpha,\beta} \left\| \mathbf{C} * \begin{bmatrix} \alpha \\ \beta \end{bmatrix} - d \right\| \tag{4-5}$$

*subject to:*

$$\left\{ \begin{bmatrix} A_{1,1:p}\alpha = x_0 \\ A_{1,1:p}\beta = y_0 \\ A_{n,1:p}\alpha = x_f \\ A_{n,1:p}\beta = y_f \end{bmatrix} \right\} \tag{4-6}$$

$$\left\{ \begin{bmatrix} \alpha_1 = t\dot{U}_x \\ \beta_1 = t\dot{U}_y \\ t > 0 \end{bmatrix} \right\} \tag{4-7}$$

Once a solution is found, the max error is defined.

$$max\_error := \max_i \sqrt{\left( \left( \sum_{j=1}^{p} \alpha_j s(i)^{j-1} \right) - x(i) \right)^2 + \left( \left( \sum_{j=1}^{p} \beta_j s(i)^{j-1} \right) - y(i) \right)^2} \tag{4-8}$$

Equation (4-8) relies on the estimate of error at a particular value of *s*, as the error due to estimating x, and the error due to estimating y.

The routine that performs the optimization code used within MATLAB is lsqlin.m. This routine is built into the optimization toolbox. It is especially built to minimize the norm of a vector subject to linear equalities and inequalities. The objective function does not minimize the maximum error, rather it minimizes the sum of the squared errors. Even though the measure of performance, maximum error, and the objective function do not match, there exist significant computational advantages in using the built-in code. The formulation of minimizing the maximum error was coded within AMPL (advanced mathematical programming language). Subsequently, we used LOQO (a non-linear programming solver) to solve the problem. With the same choice of $p$ (the degree of the two polynomials), LOQO was not guaranteed of even finding a feasible solution. However the MATLAB routines always found the optimal solution (in all the test cases). The reason for this discrepancy was likely the differences in algorithms used and the high condition number of the C matrix. In addition, the computation time for the MATLAB routine was phenomenal. Most cases (approximately 90%), took less than 0.50 seconds to find an optimal solution. For the few cases where LOQO did find an optimal solution, the time for computation was at least one order of magnitude greater than the computation time for MATLAB[1] and usually greater than two orders of magnitude in difference. The relative performance (in terms of computation time) is hardly important when considering that LOQO failed to even find solutions at times.

It should be noted that choosing the degree of the polynomial was based on experience in running the model. The degree of the polynomial was bounded between 5 and 20. The more jagged the trajectory the greater the value of $p$. Specifically, $p$ was set to two times the number of vertices in the jagged path plus three. If using this method, $p$ is larger than 20, then $p$ was set to *20*. The minimum number of vertices in the jagged path equals two (i.e. in the case of a straight path the end points of the path are considered the two vertices). Recall, that vertices in the path can be the end points of the jagged path or a point in the jagged path that connects two line segments (with different slopes). Furthermore, the number of points, $n$, was set equal ten times the number of change points in the jagged path. The total number of points was limited to 250 (clearly, if there are more than 250 change points then more points would be needed; however, this scenario never occurred in practice). In addition, the minimum number of points was limited to no less than 15. This lower limit ensured that $n$ was

[1] MATLAB computations performed on a SGI ORIGIN 2000 workstation; LOQO computations performed on MIT's Athena Dial-In Computers (similar performance as SGI machine)

sufficiently larger than *p*. As typical with most polynomial fitting schemes, one should try and ensure that *n>>p*. Otherwise, the polynomial begins to fit only at the points *[X(i),Y(i)]* (the data), and varies widely between the points.

The maximum error due to the optimization in the simulation, was rarely greater than 1 unit. In terms of the grid map, this value implies that the error due to the smoothing was usually bounded by 1 grid cell. The longer the trajectory, the greater the error. Instead of listing various runs and the data used for those runs, a sampling of the maximum error is given in the Chapter 5.

## 4.6 Algorithm Performance and Worst-Case Analysis

This section examines the properties under which a seeker will track the moving target. If the algorithm fails to track the target it is not an acceptable approach. If a measure of algorithmic performance is desired, then simulating the algorithm is one approach. However, some notion of theoretical correctness and complexity of the algorithm should not be overlooked.

### 4.6.1 Speed, and Distance Comparison

Assumptions:

1. Both Target and Seeker are Travelling in Free Space
2. Both Target and Seeker are Travelling at Constant Velocities
3. The neighborhood structure is defined as a set of different angles.

Clearly the seeker vehicle cannot track the moving target if the target travels with greater speed. However, one can make this bound tighter by examining the neighborhood structure for the seeker vehicle. First, the target travels without the restriction of performing maneuvers on a grid based map. Therefore, the target vehicle travels according to the Euclidean distance metric. Clearly, if the target vehicle is a manned submarine, the submariners on-board do not constrain their maneuvers to the 16 different moves as capable by the seeker vehicle. However, given any two points in space the distance between these points, in relation to the seeker vehicle, is not the

Euclidean distance. The question remains – *What is the equivalent metric for the seeker?* The answer is not so simple due to the grid map, and the structure of the move types. For instance, if the seeker is at the origin, (0,0) and desires to move to the point (0,1), it moves to the neighboring cell directly to the right. This move is admissible and results in a cost or distance of 1 unit. The Euclidean distance is also1. However, if the seeker wishes to move from (0,0) to (3,4), it must first move to the point (1,1), then to (2,2), and finally to (3,4). The first two moves result in 2.828 units of distance, and the last to a distance of 2.236. The total distance for the move is 5.064 units. If the target is to move from the origin to (3,4), the total distance traversed is 5 units. Therefore, some pairings of points yield the same distance, while others yield different distances. Furthermore, the concept of grid size can further effect the comparison.

- **Definition:**

  1. The set of angles in the 1st quadrant that define the possible moves is $\angle_{move} := \{0^\circ, \arctan(.5), \arctan(1), \arctan(2), 90^\circ\}$. The moves (in the 2nd quadrant) can be found by adding $90^\circ$, to these angles. The moves in other quadrants are found similarly. These angles are the equivalent angles to the moves defined by the neighborhood structure.

- **Proposition:** For any two points in the plane, the seeker may be forced to travel at most 1.0275 units for every unit traveled by the target.

- **Proof**: As before, this proof relies on some insight gained by the geometry of the problem.

  1. For any two points in space the target travels according to the Euclidean metric.

  2. Construct a circle of *radius 1* centered at the origin. If the target must travel from the center of the circle to any point on the circle, the target travels *r* units of distance

3. If the seeker is constrained to one move, only 16 different points on the circle are attainable. By symmetry, only the first quadrant need be examined. According to $\angle_{move}$, only five points are attainable on the circle in the first quadrant.
4. Geometry yields that two points in the first quadrant result in the maximum deviation from the 5 possible points. These points are shown in Figure 4.15.

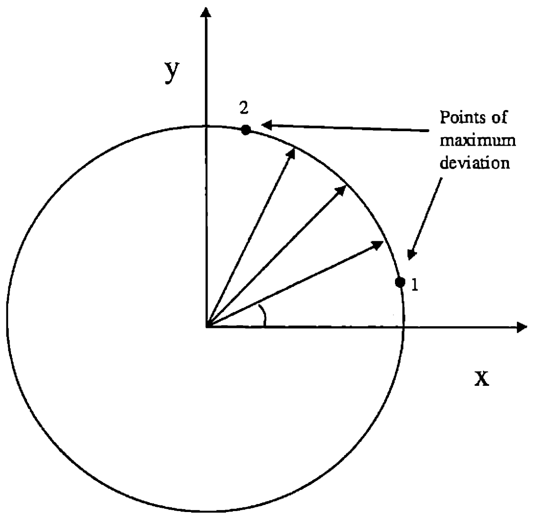


**Figure 4.15 Illustration of Angle Neighborhood**

5. By simple geometry, the points of maximum deviation occur at 13.28°, and 76.72°.
6. The seeker can attain either point of maximum deviation by performing two moves. This scenario is illustrated in the following picture.

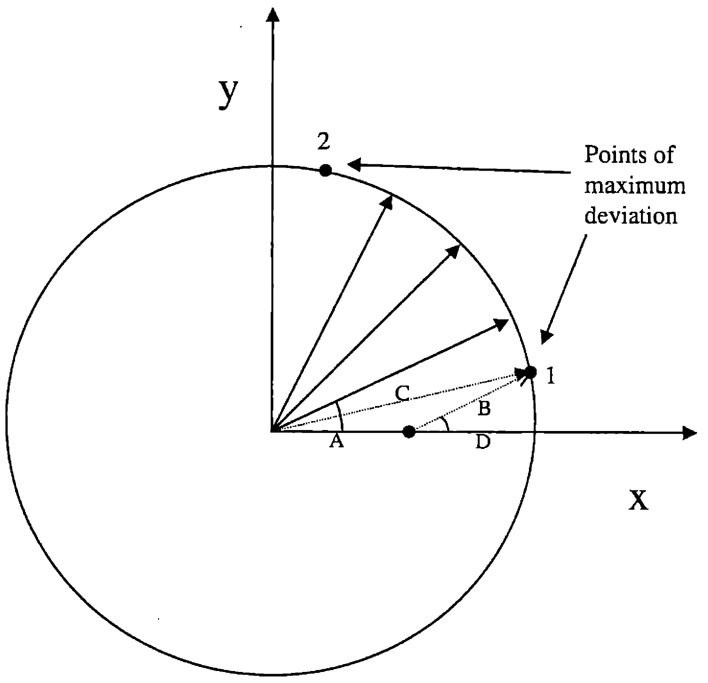


**Figure 4.16 Magnification of Angle Neighborhood**

7. Again by symmetry, the results for the point at 13.28°, *point 1,* are applicable to *point 2.*
8. In order to attain *point 1,* the seeker moves along segment *A*, and then along segment *B*.
9. Geometric insight suggests that the seeker should leave the x-axis as soon as possible in order to reach *point 1*, if the total distance to *point 1* is to be minimized. This corresponds to leaving the x-axis with a heading of approximately 26.565°, or exactly $tan^{-1}(1/2)$.
10. $\angle CA = .5*\angle BD \approx 26.565°$. Moreover, $\angle AB \approx 180 - 26.565 = 153.435°$, and $\angle BC+\angle CA+\angle AB=180$. Observe that all the angles in $\Delta ABC$ are defined, and $\Delta ABC$ is an isosceles triangle. Using the law of sines, the length of $A \approx .5137$.
11. The resulting distance traveled by the seeker is 1.0275 units. For the same two points the target vehicle traveled 1 unit.

12. If any other point on the circles is chosen to be attained by the seeker, the resulting combination of moves will have a distance less than the distance of 1.0275 units. This fact is due to the other points being strictly closer to one of the desired headings.

- **Result 1:** In the worst case, the seeker travels 1.0275 units of travel for every unit traveled by the target.
- **Result 2:** By previous arguments the seeker cannot travel slower than the target. Incorporating the distance metric result with this condition yields the following worst case condition,

$$\|V_s\| < .973\|V_T\| \tag{4-9}$$

### 4.6.2 Cycling and Worst-Case Path

At this point, a speed comparison is in place, the issue of decreasing the distance from seeker to target is discussed. In actuality, the planner as developed is concerned only with planning paths to the last known position of the vehicle. Assuming this trajectory takes some time to complete and the target vehicle continues travelling while the seeker traverses this path, the seeker vehicle will remain a certain distance from the target vehicle. Clearly, the planner is not concerned with intersecting the target. Rather, the planner is concerned with planning paths to a target location that is not the actual location of the target when the vehicle arrives at the goal of the path.

Again some assumptions are needed to motivate the following arguments. First, in order to show how the distance decreases, it is assumed that the seeker is tracking the target in an environment devoid of obstacles. Second, the worst case situation results when the target is travelling a straight line and the seeker is following directly behind the target.

#### 4.6.2.1 Cycling Argument

The no obstacle assumption is needed because of the following cycling argument. This assumption must be made because the algorithm may fail to decrease the distance to the goal in certain degenerate situations. The best method of explaining this situation is through the use of the following figure.

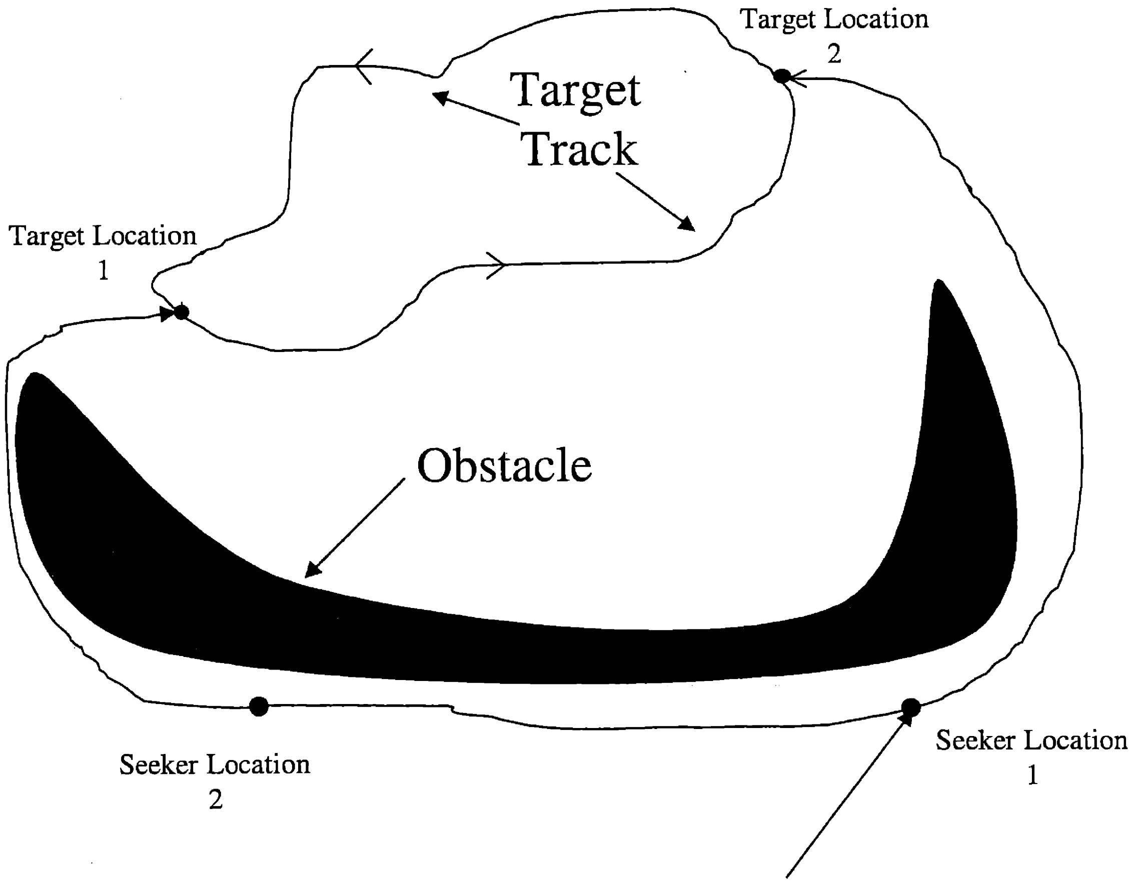


**Figure 4.17- Illustration of Cycling Behavior**

Imagine that once the seeker attains *seeker location 1*, a path to *target location 1* is available. At this point the seeker travels along that path, receives an updated target location, *target location 2.* At *seeker location 2*, a path is available to *target location 2.* This path causes the seeker to backtrack to *seeker location 1.* As the target cycles between *target location 1* and *2*, the seeker vehicle cycles between *seeker location 1* and *2.* This cycling behavior may prohibit the seeker from decreasing its distance to the target. However, for the application chosen such a situation is highly unlikely to develop. It is hard to imagine the target cycling between two locations as it tries to egress its port and travel up the coastline. Furthermore, if such a situation were to arise, the mission planner on-board the UAV could stop the cycling behavior by enforcing a plan to continue until one of the goals is reached. This behavior is not modeled as it is part of the mission planner's role and not the path planner's role.

### 4.6.2.2 Worst Case Path

The following figure describes the progression of the target to the goal.

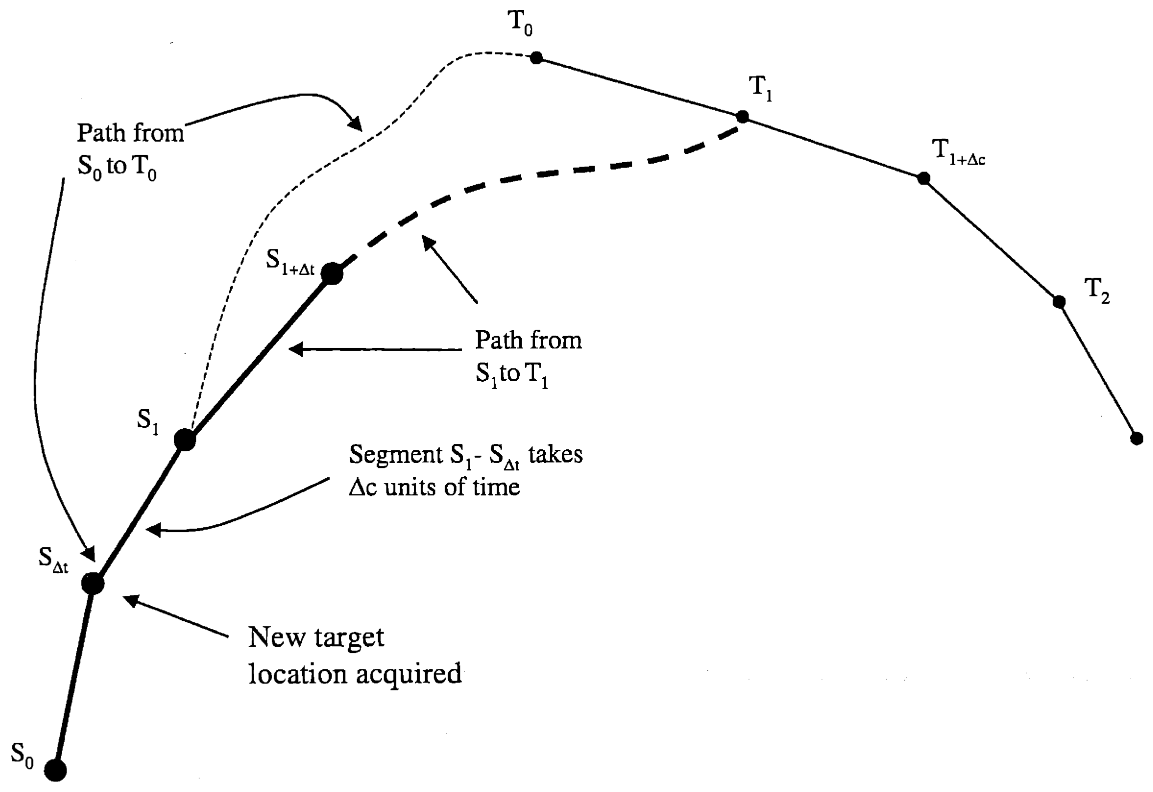


**Figure 4.18 Illustration of Path Updates**

In Figure 4.18, $S_0$ is the seeker's initial position, $S_{\Delta t}$ is the position of the seeker when the target location $T_1$ is given to the shortest path algorithm. The seeker remains on the initial path for $\Delta c$ time *($\Delta c$ = time to compute path)*, and begins travelling on the new path at $S_1$. Likewise, at $S_{1+\Delta t}$ the seeker obtains the new target location, $T_2$. If no new information is obtained, the vehicle's trajectory to $T_1$ is shown in bold. At the point $S_{1+\Delta t}$, the vehicle has traveled through the following nodes, $S_0$, $S_{\Delta t}$ & $S_1$. The path from $S_{\Delta t}$ to $S_1$ takes $\Delta c$ time to complete, where $\Delta c$ is the computation time to compute the shortest path from $T_0$ to $T_1$, and create the path from $S_1$ to $T_1$.

Figure 4.18 describes a typical series of target locations and seeker locations as the algorithm progresses. In this figure, as the seeker travels towards the target it is able to make more progress towards the target than if the target were to moving directly away from the seeker. In other words, the target will make the most progress away from the seeker if it is travelling directly away from the seeker. In this type of travel, all of the target's velocity contributes to moving away from the seeker. Therefore, this situation, where the target is travelling in a straight path directly away from the seeker, is considered the worst case path. The worst case path is shown below in Figure 4.19.

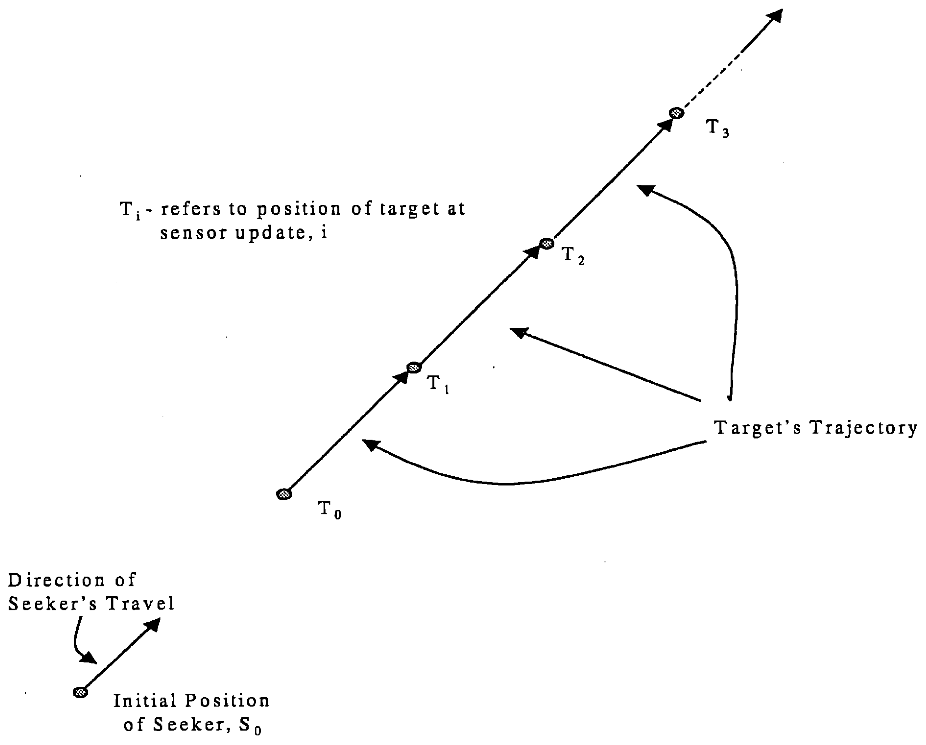

$T_i$ - refers to position of target at sensor update, i
$T_3$
$T_2$
$T_1$
$T_0$
Target's Trajectory
Direction of Seeker's Travel
Initial Position of Seeker, $S_0$

**Figure 4.19 Illustration of Worst Case Path under the No Obstacle Assumption**

In this worst case path situation, the seeker's path will also be a straight line and eventually follow directly along the target's trajectory.

### 4.6.3 Worst Case Performance

This section gives a detailed analysis of how the seeker vehicle will track the moving target under a worst case scenario. Since this analysis examines the worst case behavior, the bound shown to compare the two velocities should also be used.

Throughout this section this bound is not included, but is implicit whenever the velocities are used. Specifically, wherever $\|V_T\|$ is used, it can be thought of as$1.0275\|V_T\|$. . In order to motivate this analysis a few conditions are needed. Some of these are listed previously and are reiterated in the following list.

1) The network is a 2-D grid map.

2) The magnitude of the seeker's velocity, $\|V_S\|$, is assumed constant as is magnitude of the target velocity, $\|V_T\|$. In addition, $\|V_S\| > \|V_T\|$.

3) The target locations are passed to the path planner at a fixe rate, Δt.

4) The computation time is bounded by *Δc*. Furthermore, *Δc* is smaller than *Δt*.

5) The seeker is tracking a target travelling along the worst case path.

6) There exist no obstacles in the environment of operation.

7) If the seeker reaches the goal of the current path and no new path information is available (i.e. seeker vehicle is in the terminal state), the seeker vehicle waits at this goal until new trajectory information is available. Once the new trajectory is available the seeker begins travel instantaneously at$\|V_S\|$.

8) The smoothing of the path results in the actual straight-line path that the seeker must follow in the worst case path scenario.

9) The distance from the target's initial location to the seeker's initial location is defined to be larger than the amount of distance the seeker can travel in a Δ*t*+*Δc* time period. Defining the initial distance in this manner provides insight into how the algorithm performs when the initial distance is large.

After all these conditions are established a sense of how the algorithm performs can be developed. The majority of this analysis will focus on the improvements made to the initial distance separating the two vehicles at the start of the mission.

- **Proposition:** If the target and seeker are in the worst case path situation and condition #9 (from previous list) is satisfied, then the decrease in the distance from the seeker to target during one iteration is $\Delta d_{improve} = \left(\|V_s\| - \|V_T\|\right)\left(\Delta t + \Delta c\right)$.

- **Proof:**

  1. The initial distance is defined,

$$d_0 = \|T_0 - S_0\| \tag{4-10}$$

  2. After on step of the algorithm the distance is

$$d_1 = d_0 - \|V_S\|(\Delta t + \Delta c) + \|V_T\|(\Delta t + \Delta c) \tag{4-11}$$

     Both the seeker and target can travel for *Δt + Δc* due to the statement in the proposition.

  3. The improvement in the distance is

$$\Delta d_{improve} = \left(\|V_s\| - \|V_T\|\right)\left(\Delta t + \Delta c\right) \tag{4-12}$$

- **Result:** With the conditions set up in the proposition, the seeker gains on the target at each iteration by, $\Delta d_{improve} = \left(\|V_s\| - \|V_T\|\right)\left(\Delta t + \Delta c\right)$.

Now, if the distance continues to decrease at some point the distance is small enough that the seeker can no longer travel for the entire *Δt+Δc* time. However, the target still travels for this length of time. If the amount of distance traversed by the seeker is too small then the target vehicle may begin to pull away from the seeker. Let the amount of time that the seeker is able to travel be defined as Δx. Now, the improvement is given as,

$$\Delta d_{improve} = \|V_S\|(\Delta x) - \|V_T\|\left(\Delta t + \Delta c\right) \tag{4-13}$$

If the seeker is to decrease the distance to the target, then the improvement must be positive. In other words, the following condition must hold,

$$\frac{\|V_S\|}{\|V_T\|} > \frac{(\Delta t + \Delta c)}{\Delta x} \tag{4-14}$$

Since, all the parameters in (4-14), except *Δx* are fixed for an entire mission, insight can be gained by looking at how small *Δx* can become. The smallest *Δx* value will result from the shortest possible path that the seeker may encounter. This situation results when the seeker reaches the goal of the current path. Upon reaching the goal of the current path, the seeker must wait for the next target update (if not already provided), and wait for the computation to complete. Now the vehicle, is ready to travel to the next target location. Since the target locations are equally spaced in time and space (i.e. due to the target's constant speed), the distance for the next path is known. This distance will correspond to the smallest value for *Δx*. Namely,

$$\Delta x_{min} = \frac{\|V_T\| \Delta t}{\|V_S\|} \tag{4-15}$$

- **Proposition:** If the seeker travels along the minimum distance path, the distance from seeker to target will increase.
- **Proof:** The following proof is simple.

1. The minimum distance path occurs when the seeker travels from a target location to the next target location. Again,

$$\Delta x_{min} = \frac{\|V_T\| \Delta t}{\|V_S\|} \tag{4-16}$$

2. Substituting into (4-14),

$$\frac{\|V_S\|}{\|V_T\|} > \frac{(\Delta t + \Delta c)}{\frac{\|V_T\| \Delta t}{\|V_S\|}} \tag{4-17}$$

3. Rearranging (4-17),

$$\Delta t > \Delta t + \Delta c \tag{4-18}$$

4. The condition for improvement is violated, Thereby, using (4-13) the growth in distance is given by

$$\Delta d_{improve} = \|V_s\| \frac{\|V_T\| \Delta t}{\|V_S\|} - \|V_T\| (\Delta t + \Delta c),$$
$$\Delta d_{improve} = -\|V_T\| \Delta c \tag{4-19}$$

- **Result**: If the seeker travels along the minimum distance path the distance grows according to (4-19).

*What does this result imply?* After the seeker closes the initial distance and reaches the goal of a current path, the seeker is not guaranteed of improving the distance to the seeker. In addition, once the distance is closed such that the seeker reaches the goal of the current path the seeker will reach the goal of all subsequent paths as inferred by (4-16).

This result demonstrates that in the worst case scenario as developed the distance from seeker to target grows once the seeker closes the initial distance as is able to reach the goal of the current path. This situation may only arise after many iterations of the path-planning algorithm. However, once the goal of a current path is reached the distance begins to increase at each iteration. This result seems counterintuitive, however it is possible. (4-19) implies that the amount of gain the seeker is able to make when travelling from goal location to goal location is unable to counteract the gains made by the target.

The reason for the poor performance of this algorithm is the fixed rate of the target location updates. The fixed *Δ*t forces the seeker to use *Δt +Δc* time units at each iteration of the algorithm, of which some may be spent waiting for new path information. After the seeker closes the initial distance and reaches the goal of the current path, the seeker is ensured of reaching the goal locations of all subsequent paths. However, now the seeker ends up waiting at the goal nodes too long, such that it cannot gain on the target.

All hope is not lost in developing an approach that can maintain the distance improvement at each iteration. The framework as presented earlier in this chapter is adaptable to variable target location updates. Seeing as how this condition was the driving factor for the poor worst case performance, incorporating a variable $\Delta t$ approach may alleviate the problem. In addition, the terminal state condition as dealt with in this section enforced the seeker to wait at the current node until new information became available. A variable $\Delta t$ approach may reduce the likelihood of such a situation by requesting a target location, at least $\Delta c$ time units away from the goal of the current path. In this manner, once the seeker reaches the current goal, a path already exists and no waiting is needed. The variable $\Delta t$ approach is developed in the Chapter 5. The approach is not specified here because the error growth as suggested by this analysis was only seen after running the simulation.

If the vehicle is forced to use a fixed $\Delta t$, a different approach is provided. This approach relies on a heuristic that guesses the position of the target some time in the future. In this manner getting good estimates of future positions of the target hopefully eliminates the growth in distance. Again, this method is presented in Chapter 5.

In summary, the path planner as analyzed with a fixed $\Delta$t has some interesting performance characteristics. First, the initial distance is closed at a rate of $\|V_S\| - \|V_T\|$. As long as the current path is longer than $\|V_S\| (\Delta t + \Delta c)$, then the distance will improve at this rate. As this distance decreases, the seeker vehicle will eventually reach the goal of a current path and wait for new trajectory information. At this point, the target vehicle moves away from the seeker vehicle. The algorithm performs well at closing the initial distance but performs poorly after this distance is closed. Simulation results in the following chapter show this exact conclusion.

## 4.7 Heading Command Extraction

After a path is found and smoothed, aside from the 2-turn section, it can be stored compactly as the coefficients of the polynomials describing the $(x,y)$ location of any point along that trajectory. This section deals with methods for extracting needed control commands for the vehicle.

Many control systems on-board UAVs are designed to handle a table of vehicle heading commands indexed through time. The heading commands needed are not ground track headings but the heading of the vehicle. These vehicle headings are sent to the control system and executed. In the terms of a UUV, the heading commands are differentiated with respect to time to give a heading rate. This heading rate then translates to a rudder position. The details of how the control system translates a heading into a rudder position is not provided and is considered a black box within the UUVs architecture.

In the presence of current, the vehicle adjusts its heading to counteract the effects of current. In this setting, current only effects the velocity of the vehicle (i.e. only a first-order effect). The approach taken in this section uses discrete approximations of the ground track to find the heading of the vehicle through time. Essentially, as the vehicle progresses along the ground track it uses the most up to date estimate of the current to adjust the heading of the vehicle. The vehicle must align the body velocity vector, $V_b$ such that the current vector, $c$, plus the body velocity, $V_b$, vector aligns with the ground track heading vector. The vector that results from the addition of $c$ and $V_b$ is the ground track velocity vector, $V_g$. This vector is then used to simulate the trajectory in time. The following figure illustrates the interrelationship between these velocity vectors.

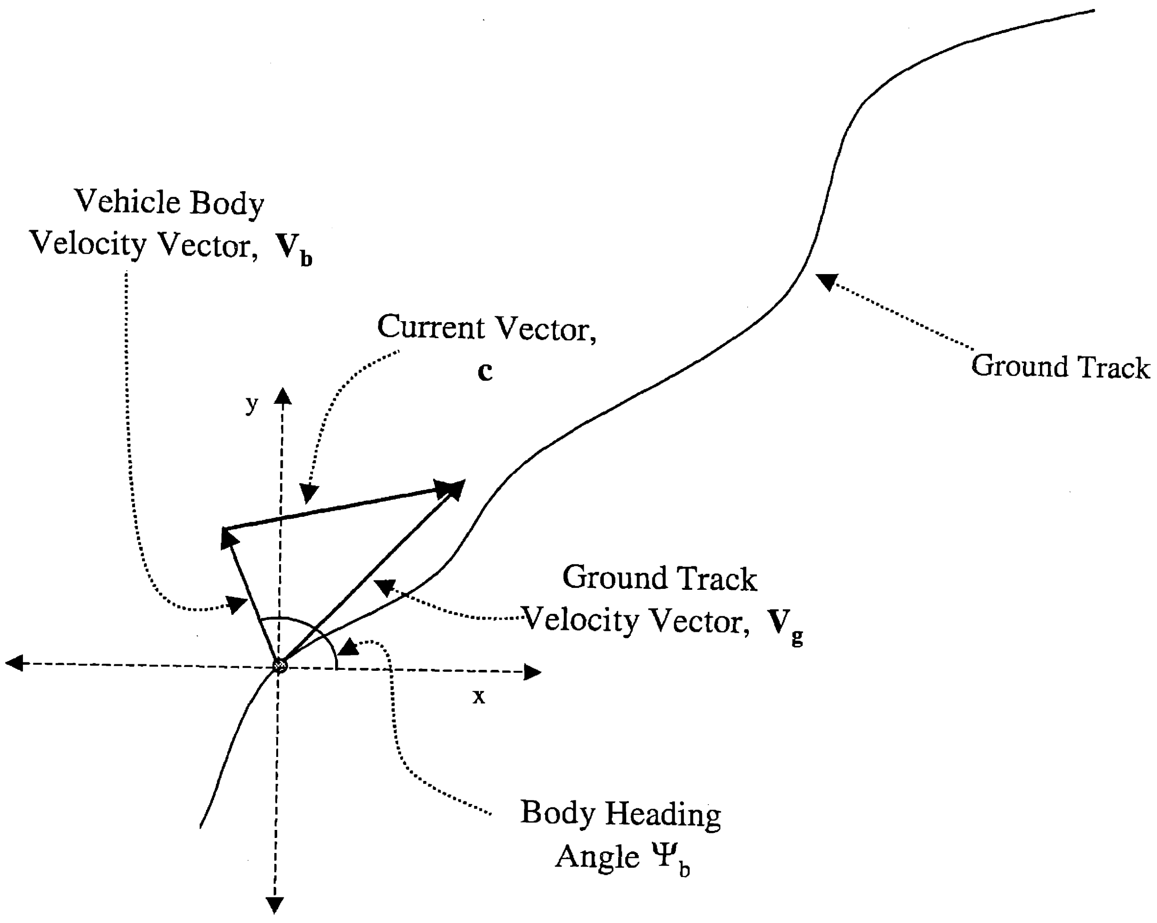


**Figure 4.20 Illustration of Heading Command Extraction**

In Figure 4.20 certain parameters of each vector are known for a specific point on the trajectory. The current vector, *c*, is fixed. The magnitude of $V_b$ is fixed, and the unit vector in the direction of $V_g$ is known (in the path smoothing chapter this is referred to as the unit tangent vector). The needed heading is $\Psi_b$. Simple trigonometry yields the correct value for $\Psi_b$. In order to simulate the trajectory in time the more important information is $V_g$.

For a specific value of *s* (see Chapter 3), both *x* and *y* values can be found. The algorithm can loop over all values of *s* for which the current trajectory is defined or terminated at a specific time (i.e. the entire trajectory is not needed). The heading extraction algorithm now follows:

```
**algorithm** *Heading Commands;*
initialize $\Delta s$;
initialize $t$ {$= t_f$ of previous trajectory};
$s = 0$;
**while** $(s \le s_f - \Delta s) \& (t \le t_f)$
    $P_1 = \langle f(s), g(s) \rangle$
    $P_2 = \langle f(s+\Delta s), g(s+\Delta s) \rangle$
    compute $T(s)$ {unit tangent vector of ground track};
    get $c$ {best estimate of current at time $= t$};
    compute $V_g, V_b, \Psi_b$;
    $t = t + \frac{\|P_2 - P_1\|}{\|V_g\|}$ { $\|P_2 - P_1\|$ is estimate of arc length at $s$};
    $s = s + \Delta s$;
    append $\langle t, \Psi_b \rangle$ to control commands;
**end**;
```

**Figure 4.21 Heading Command Extraction Algorithm**

The heading command algorithm can run in real-time as the UUV traverses the current path. In this manner, the most current estimate of the underwater current can be used. However, the algorithm can be run ahead of time and the heading commands stored for use later by the UUV.

This algorithm was presented for the part of the trajectory defined by the two polynomials. However, a part of the trajectory namely the 2-turn segment is not mentioned in this discussion. The heading commands for this portion of the trajectory can be found using the same approach. Instead of functions describing the points along the 2-turn segment, a list of these points can be stored as output from the 2-turn algorithm. Then the algorithm as shown in Figure 4.21 applies, where *T(s)* is found from the first difference of adjacent points along the 2-turn segment.

## 4.8 Implementation Overview

The entire simulation was coded within a MATLAB environment. The MATLAB code serves as the backbone of the path planner and maintains all the needed data for the shortest path calculations. The specific algorithm chosen for computing the shortest path algorithm is the D* algorithm as developed by Stentz [15], and implemented at Draper Laboratory, under funding from the Naval Underwater Warfare Center (NUWC). The code is written in C++, and is run on Silicon Graphics workstation (as is the entire path planner simulation). A step by step description of the path planner is given below:

1) Initialization – Initialize all needed problem variables and parameters. (Vehicle speeds, map, target track, initial position of seeker, time...)

2) Get source and goal locations.

3) Find Shortest Path

4) Perform 2-Turn Procedure if needed

5) Smooth Path (constrained model w/o curvature constraints)

6) Step along new smooth trajectory until goal is reached or new target location is requested

7) If new target information is requested go to step 2

8) If goal is reached wait until a new target location is requested, then go to step2.

The above description is a gross over simplification of the code. Some of these steps must be carried out carefully to correctly model the simultaneous functionality of an actual vehicle. For instance, in step 8 the vehicle actual acquires the target location and computes the shortest path while waiting at the current node. When a target request occurs while the seeker is travelling along a trajectory, the seeker continues travelling along that trajectory until the shortest path code and smoothing complete. However, the seeker can only continue along that trajectory for as long as the trajectory is defined (i.e. the trajectory ends at the goal of the current path).

To describe further the code, a more detailed look at some specifics are provided. The MATLAB code begins by passing the needed source and target information to the shortest path code. Once complete the needed information, including the backpointer information is stored within MATLAB. The path is then smoothed using optimization routines built-in to MATLAB. From this point the trajectory is simulated in time, and the vehicle begins its progression along this trajectory. After $\Delta t$ time units has expired (or a sensor update is requested), the target's most current position is passed to the shortest path algorithm. The simulation is then paused, and awaits the return of the shortest path. Once the shortest path algorithm completes, the simulation continues. The vehicle continues along the old trajectory for $\Delta c$ units of time. Now, the path is found from current position to target location. The path is pieced using the 2-turn procedure, smoothed, and then the simulation continues. Finding the path, implementing the 2-turn procedure, and smoothing the path must all occur in near real-time. Finding the path is an extremely quick operation as it is composed of following a set of pointers (i.e. backpointers) from the current node to the goal. The 2-turn procedure is also an extremely fast procedure as it is essentially a closed form solution. If the smoothing takes too long, it can be computed while the vehicle travels along the 2-turn solution. Now, the simulation steps repeat.

The methods used in the simulation make some assumptions about the vehicle's performance. First, the vehicle is assumed to perfectly track a smoothed trajectory. In reality, this assumption will not hold. Many factors (e.g. sensor error, current estimate error, inertial navigation error, etc.) may invalidate this assumption. However, it is assumed that the vehicle is equipped with a real-time controller that is capable of correcting for any deviations from a desired path. The controller will not be able to maintain the path exactly, but the assumption is made that the controller works well enough to create a simulation that disregards this issue. Second, the curvature constraint is ignored and any path generated will hopefully not violate this constraint. If after many simulation runs the curvature constraint is never violated it need not be included in smoothing procedure. However, if the curvature is violated, then the vehicle controller is assumed to handle this trajectory in an efficient manner. Finally, the time for computation is bounded by some value $\Delta c$. This assumption allows the path computation time to be a parameter of the model. Treating the computation time in this manner allows the algorithm to be analyzed at differing levels of computational power.

# Chapter 5

# Path Planner Performance

This chapter details the performance of the path planner on a select number of test cases. All of the test cases are performed on the same map as shown in the previous chapter. The test cases are chosen to represent typical track and trail type missions as defined in the *Navy Unmanned Undersea Vehicle Master Plan* [5]. According to this document, the typical mission would involve a UUV acquiring a target as it leaves port, and then proceed by trailing the target. As implemented, this type of operation is simulated by a target trajectory that leaves its home port (i.e. within the bay portion of the map) and continues along the coastline for the remainder of its trajectory. If the seeker's trajectory is largely along the coastline, then the seeker will operate in a less obstacle dense environment, than if the seeker were to perform a majority of its mission within the bay region. However, the planner is also tested under a few different scenarios, where the target maneuvers solely within the bay portion of the map.

The majority of results focus on the tracking ability of the seeker, and not on the performance of the path smoothing function of the path planner. The performance of the path smoothing is given briefly in order to indicate the accurateness of the smoothing techniques. In the context of the test cases, the obstacles are assumed buffered by a sufficient amount, such that any incursion of the trajectory with an obstacle on the map does not imply a real collision.

## 5.1 Description of Test Cases

The mix of problem parameters defines a test case. These parameters include the following:

1) Trajectory of the Target

2) Initial Position of the Seeker

3) $\|V_s\|, \& \|V_T\|$ (magnitude of seeker and target velocities)

4) $\Delta t, \& \Delta c$ (target location update rate, and time allowed for computation)

In order to create test cases, a few different values or choices were made for each parameter. Three different target trajectories were selected and labeled, $Target_1, Target_2, \& Target_3$. The first two trajectories begin in the bay and proceed out of the bay and along the coastline. The other trajectory stays within the bay for its entirety. Three initial positions of the seeker were selected and labeled $S_1, S_2 \& S_3$. The magnitude of the seeker vehicle was fixed for all test cases, and only the target's velocity was allowed to change. For all of the test cases $\|V_S\| = 18$ knots. With the given $\|V_S\|$, the target's velocity was chosen as a percentage of $\|V_S\|$. Five percentage levels, 200%, 100%, 90%, 75%, and 50%, were selected. The 200% choice is selected for only one test case and demonstrates the target moving away from the seeker. $\Delta t$ was chosen at levels of 200,150,and 100 seconds. Finally, the following times were selected for $\Delta c$: 6, 3, and 0 seconds. In total 18 sample cases are shown and are summarized in the following table,

| Test Case | Target Trajectory | Initial Seeker Position | $\|\|V_T\|\|$ | Δc | Δt |
|---|---|---|---|---|---|
| 1 | 1 | 1 | 2.0 | 3 | 200 |
| 2 | 1 | 1 | .8 | 3 | 200 |
| 3 | 2 | 1 | .6 | 3 | 200 |
| 4 | 2 | 1 | .4 | 3 | 200 |
| 5 | 1 | 3 | .8 | 3 | 200 |
| 6 | 1 | 3 | .6 | 3 | 200 |
| 7 | 1 | 3 | .4 | 3 | 150 |
| 8 | 2 | 1 | .6 | 3 | 150 |
| 9 | 2 | 1 | .6 | 3 | 100 |
| 10 | 3 | 2 | .6 | 3 | 150 |
| 11 | 3 | 2 | .6 | 3 | 100 |
| 12 | 1 | 1 | .6 | 3 | 200 |

| | | | | | |
|---|---|---|---|---|---|
| 13 | 1 | 1 | .6 | 3 | 150 |
| 14 | 1 | 1 | .6 | 6 | 100 |
| 15 | 2 | 2 | .4 | 6 | 150 |
| 16 | 2 | 2 | .6 | 6 | 200 |
| 17 | 2 | 2 | .4 | 0 | 150 |
| 18 | 2 | 2 | 1 | 0 | 150 |

The test cases are not generated as a result of all possible combinations of the problem parameters. Rather, a representative sampling of these combinations are chosen. Furthermore, a sufficient difference in $\Delta t$, and $\Delta c$ was maintained such that the seeker is not overburdened with performing computations relative to any progress it may make on the target vehicle. In addition, the $\Delta c$ times of 6 and 3 seconds were reasonable times in the face of actual computation times. Actual computations of the shortest paths required between 2.5 - 4.5 seconds (in a vast majority of cases). Finally, most of the test cases were allowed to run for approximately 3,000 seconds.

The three target trajectories are shown in the following figures.

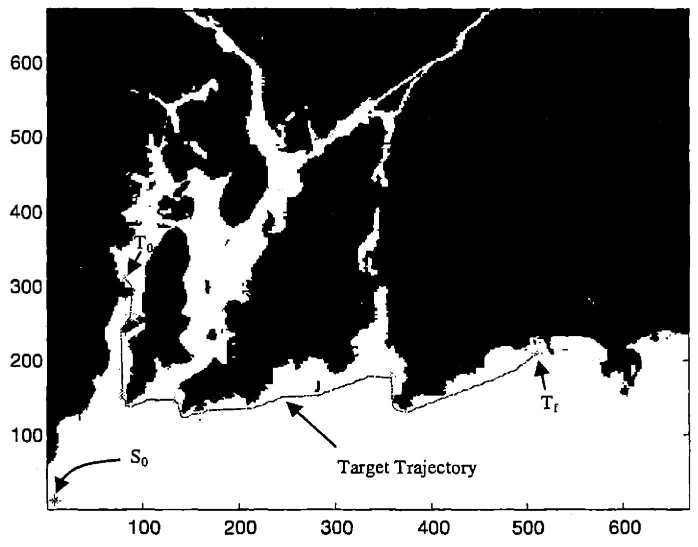


**Figure 5.1 Illustration of Target$_1$ and S$_1$**

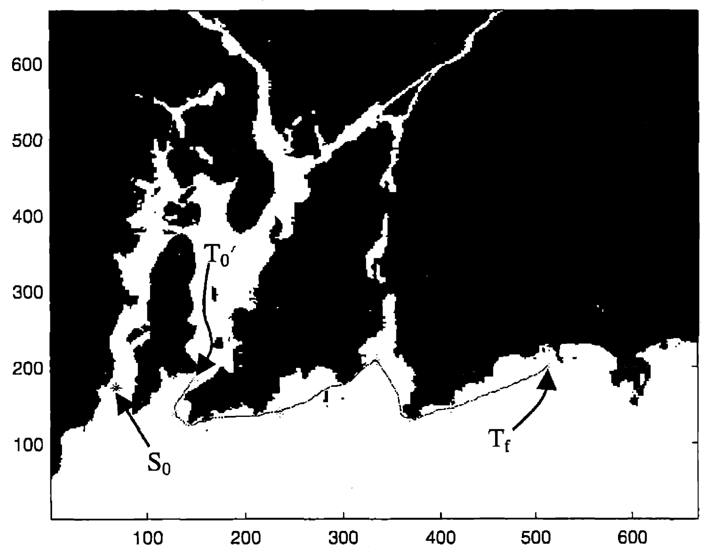


**Figure 5.2 Illustration of Target$_2$ and S$_2$**

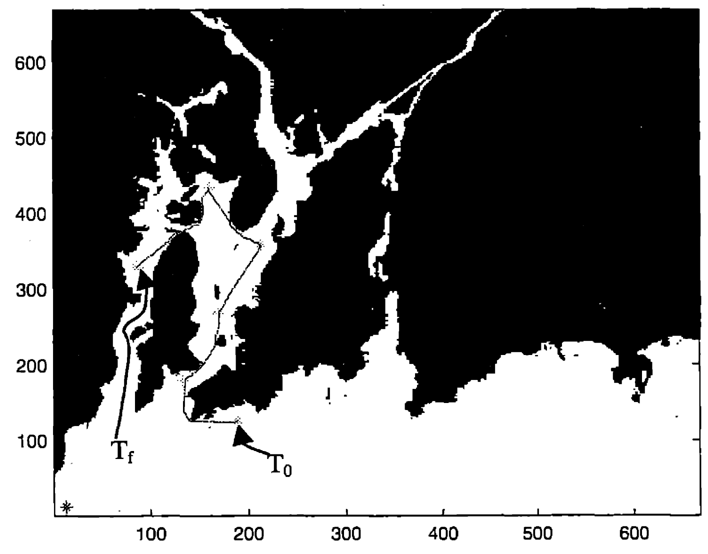


**Figure 5.3 Illustration of Target$_3$ and Seeker Init$_1$**

## 5.2 Analysis of Test Cases

The results for the following test cases are summarized in the following subsections. For this entire section, the target's position was sensed at fixed intervals in time. No propagation of where the target might go is included in this analysis. All of the test cases in this section use a fixed $\Delta t$ representation. In addition, when the terminal state exists, the seeker waits at the current node until a new trajectory is available. According to the previous chapter, if the seeker can close the initial distance to the target such that the seeker reaches the goal of each path, the error at each iteration will grow like $||V_T||\Delta c$ in the worst case. This error propagation was found using a worst case analysis, and hopefully will not be experienced in practice. For most of the test cases two figures are provided. The first figure illustrates the track the seeker took in order to follow the target's trajectory. The second figure illustrates the Euclidean distance between the target and seeker versus time. These distances are given in grid cell units (50 meters). For some of the test cases, a magnification of one of the figures is included to illustrate a specific point. Test Cases 13-18 give just the error plot, and not the ground track plot. The ground track plots are excluded because previous test cases illustrate sufficiently how typical ground tracks appear

In the discussion of test case #9, further information about the test case is provided. The smoothing error for this case is illustrated in order to validate that the smoothing is effective. The same data is present for all the other cases but in the interest of too much information, the data is only presented for this specific test case. The performance of the smoothing for the other test case is similar and can be inferred by looking closely at the target track figures. It should be noted that the smoothing technique over all test cases never produced an error greater than 2.0 grid cells.

### 5.2.1 Test Case #1

The following picture, Figure 5.4, illustrates the trajectory of the target and seeker. This type of plot is used for the remaining test cases. As with most trajectories of the seeker, after some time has expired this seeker trajectory tends to follow the trajectory of the target trajectory. The remaining illustrations will not have the labeling as present in this figure. However, the shading present in this figure, is the same for the following test cases and should serve as a means to discern the two trajectories.

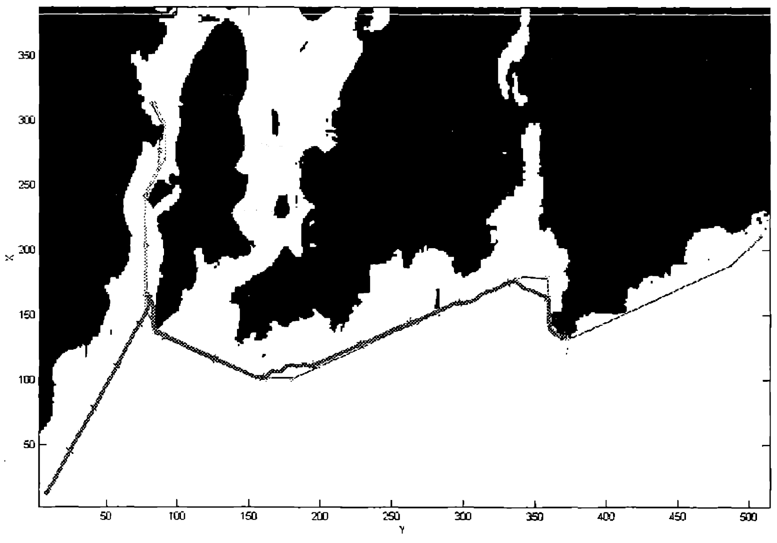


**Figure 5.4 Test Case #1 Track**

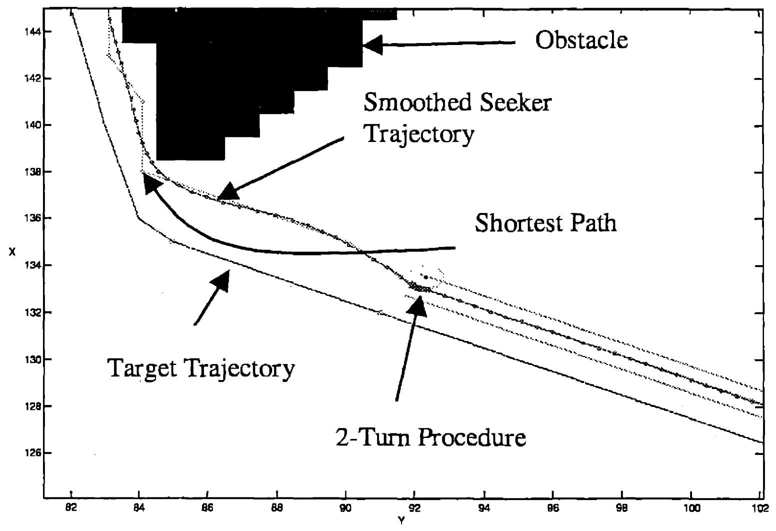


**Figure 5.5 Magnification of Trajectory Information**

Figure 5.5 is a magnification of Figure 5.4. It illustrates the shortest path, both trajectories, and the 2-turn procedure. No insight should be gained from this picture. Rather it is just an illustration of the algorithm in action.

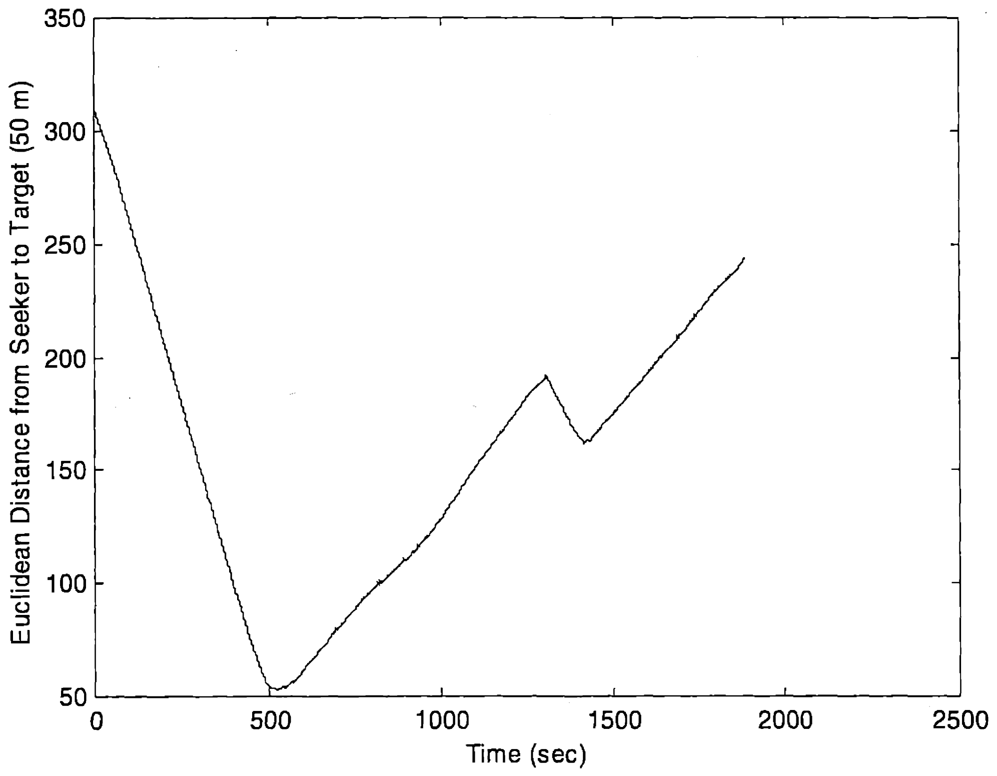


**Figure 5.6 Test Case#1 Distance Plot**

Figure 5.6 illustrates the distance from the seeker to target versus time. Clearly, the distance from seeker to target closes rapidly as the two vehicles approach each other. However, as time progresses the distance grows as expected (i.e. the target is travelling twice as fast as the seeker).

## 5.2.2 Test Case #2

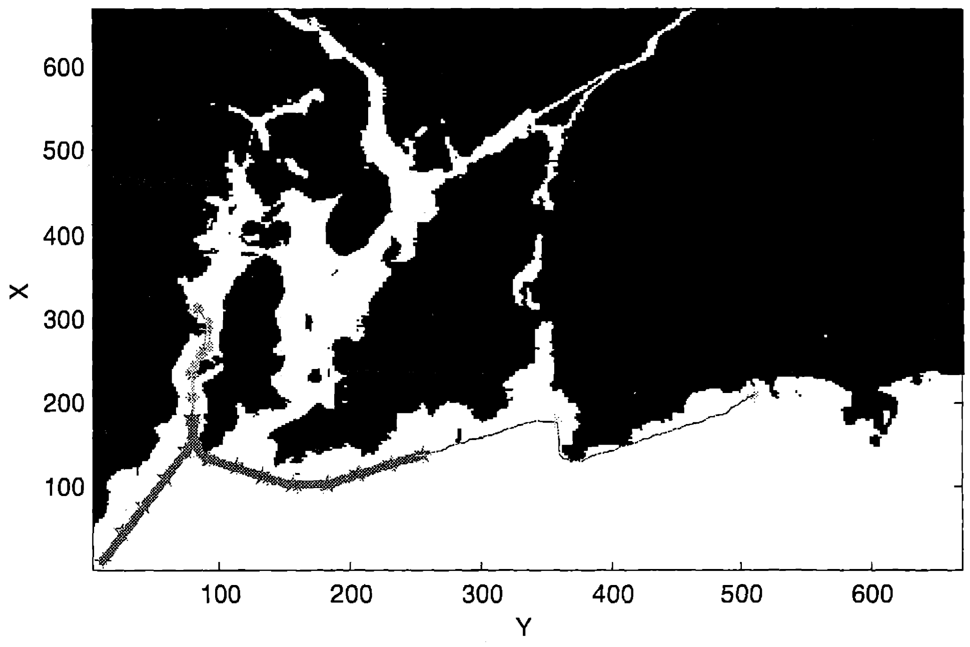


**Figure 5.7 Test Case #2 Track**

Notice that in Figure 5.7 the trajectory of the seeker vehicle actually looped back on itself, as it began to head into the bay. In this example, the seeker was able to track the target as expected.

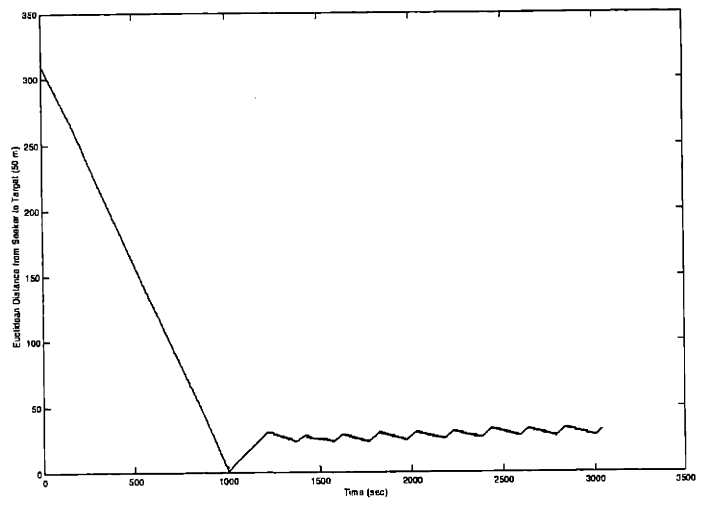


**Figure 5.8 Test Case #2 Error Plot**

Figure 5.8 shows how the distance from seeker to target closes rapidly, at first. This is a function primarily of the geometry of the problem. After approximately 1000 seconds the seeker is in a true trail type mission. The distance does not strictly decrease because the seeker frequently arrives at the goal of the current trajectory before a new trajectory is ready. Previously defined as the terminal state condition, this condition frequently increases the distance from seeker to target. However, this error is at least partially counteracted by the progress made toward the target at each iteration. As will be seen with most of the other plots, after the initial distance is closed, the distance from the target seems to oscillate. Again, this occurs when the seeker travels to the goal of the current path, and waits at that goal location until a new target location is available. The target is able to pull away from the seeker while the seeker waits for the next $\Delta t + \Delta c$ to expire (i.e. the seeker must wait for both the target location and the path for that target location).

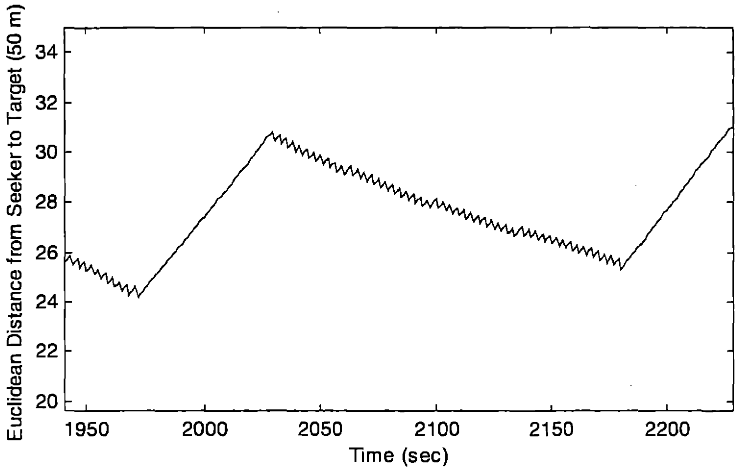


**Figure 5.9 Magnification of Error Plot**

Figure 5.9 is provided to illustrate the terminal state condition. At approximately 1975 seconds the seeker is approximately 24 x 50 meters from the target. For the next 50 seconds the distance grows linearly through time. This corresponds to the seeker waiting at a node while the next path and trajectory are created.

### 5.2.3 Test Case #3

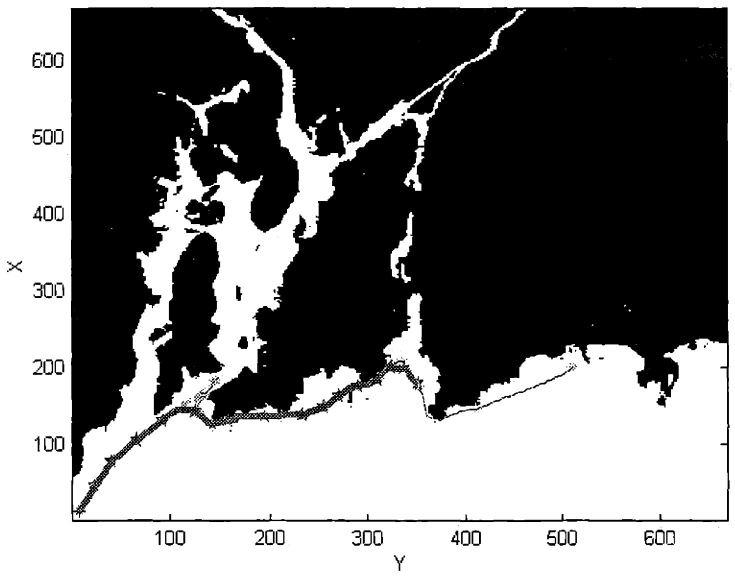


**Figure 5.10 Test Case #3 Track**

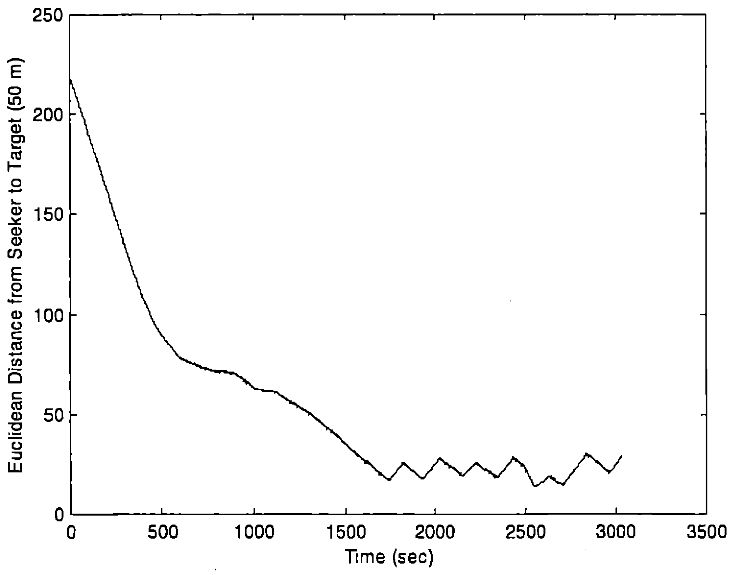


**Figure 5.11 Test Case #3 Error Plot**

### 5.2.4 Test Case #4

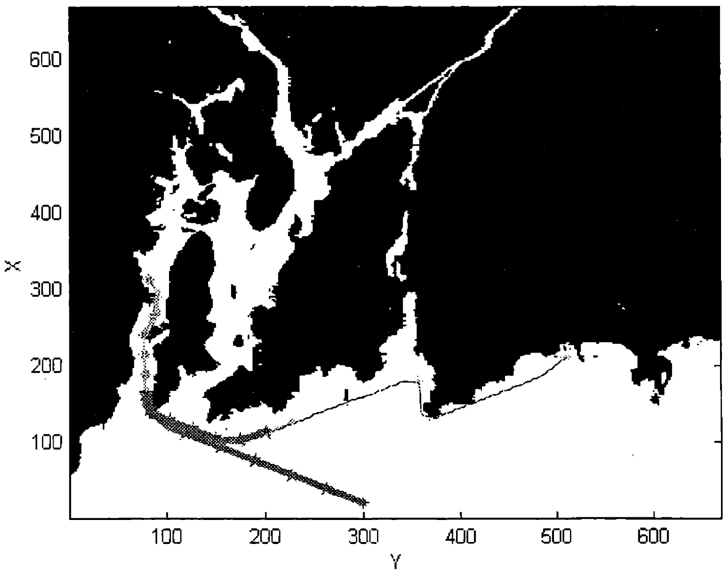


**Figure 5.12 Test Case #4 Track**

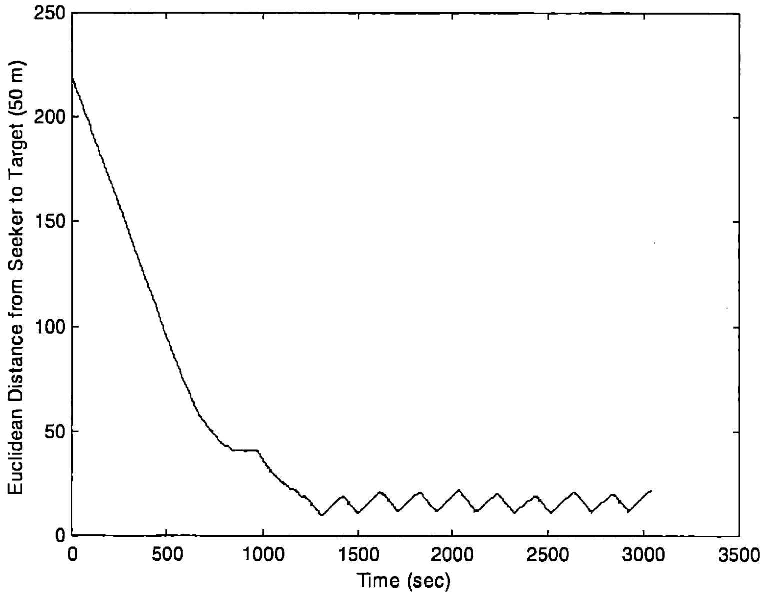


**Figure 5.13 Test Case #4 Error Plot**

## 5.2.5 Test Case #5

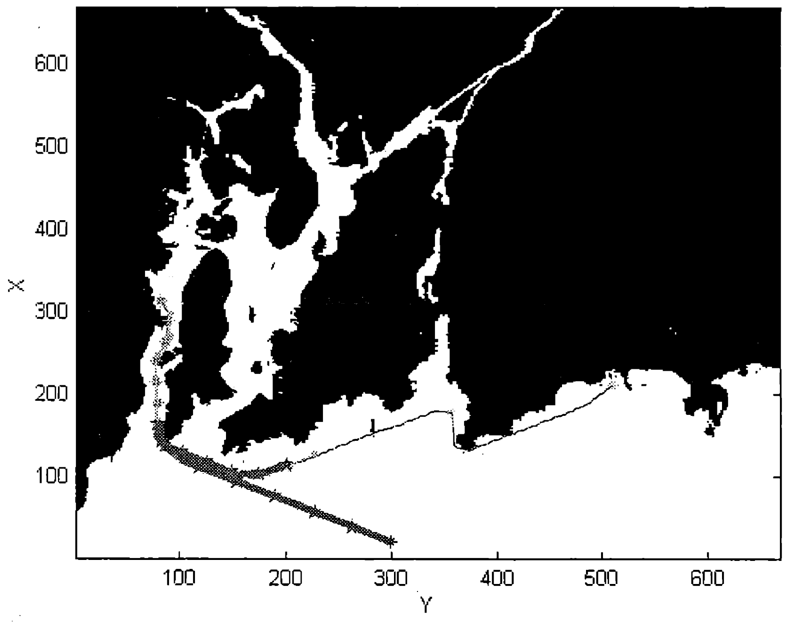


**Figure 5.14 Test Case #5 Track**

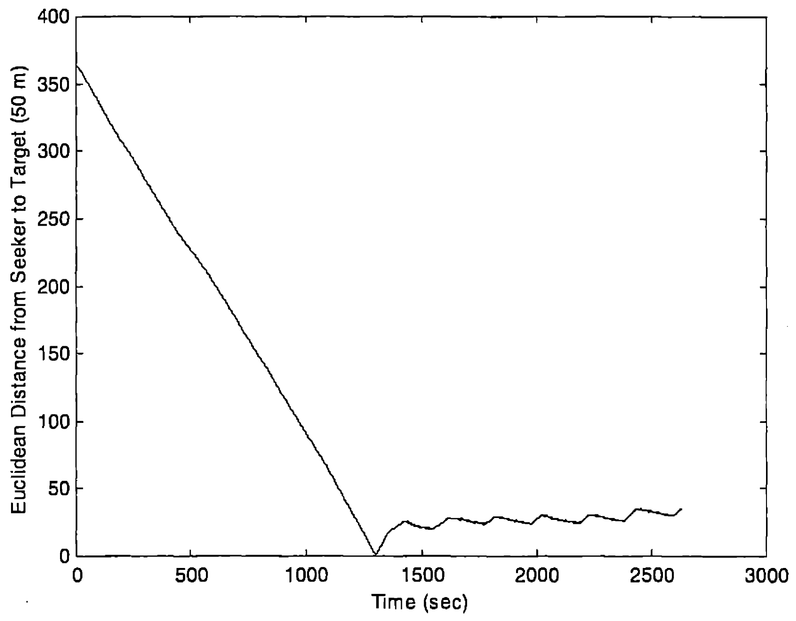


**Figure 5.15 Test Case #5 Error Plot**

## 5.2.6 Test Case #6

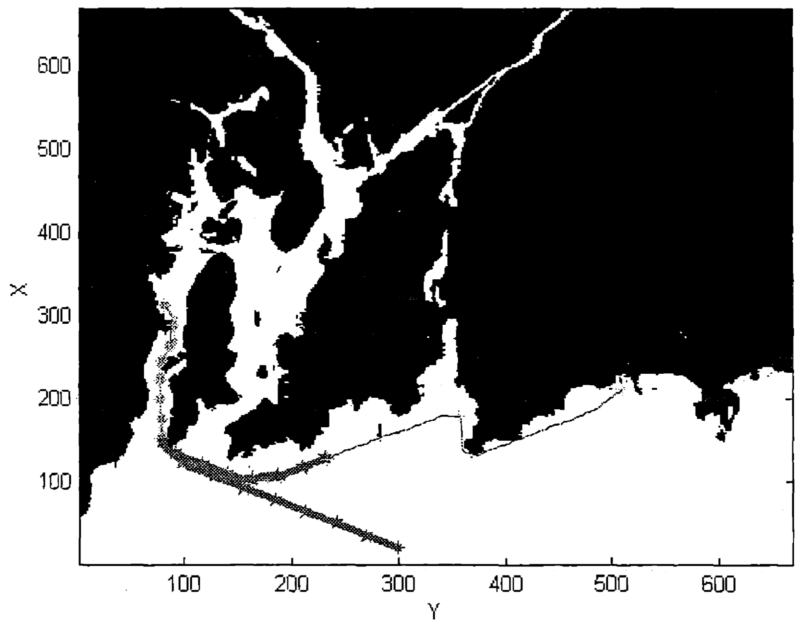


**Figure 5.16 Test Case #6 Track**

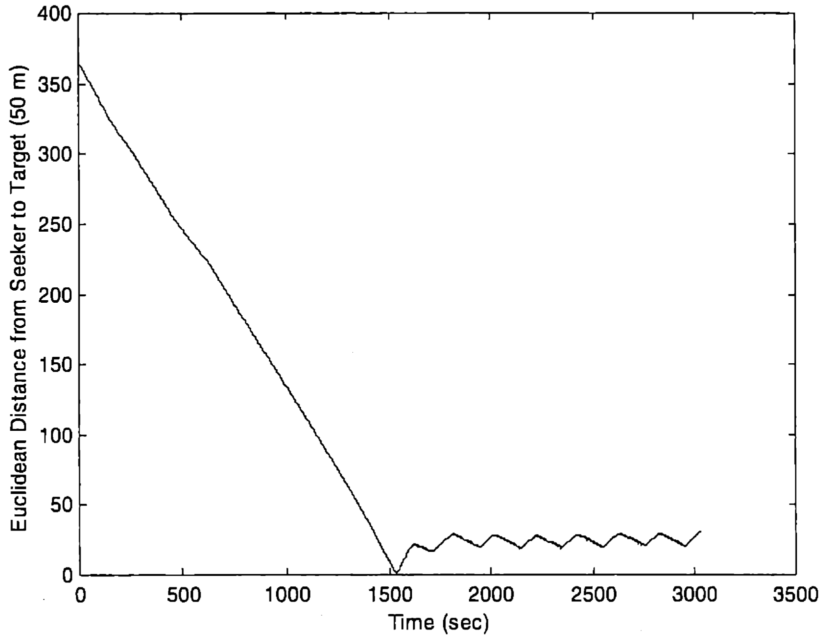


**Figure 5.17 Test Case #6 Error Plot**

### 5.2.7 Test Case #7

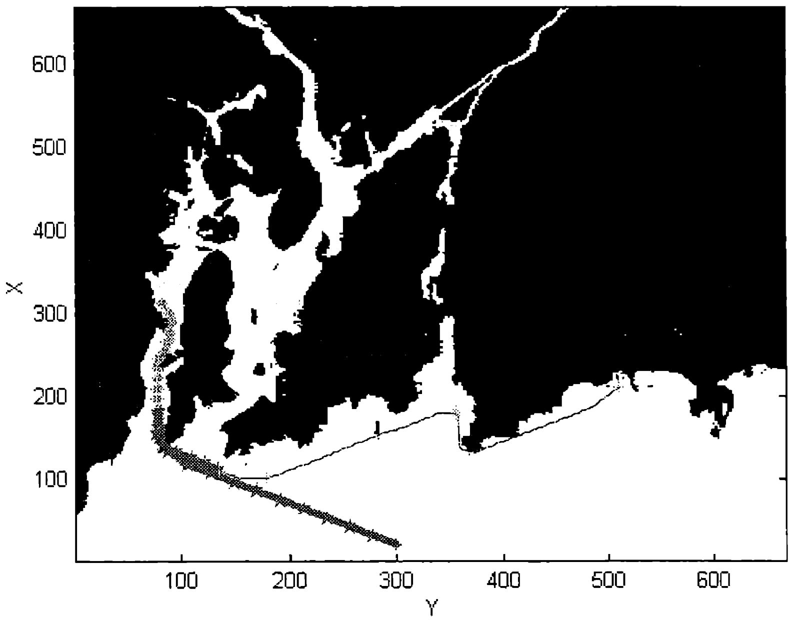


**Figure 5.18 Test Case #7 Track**

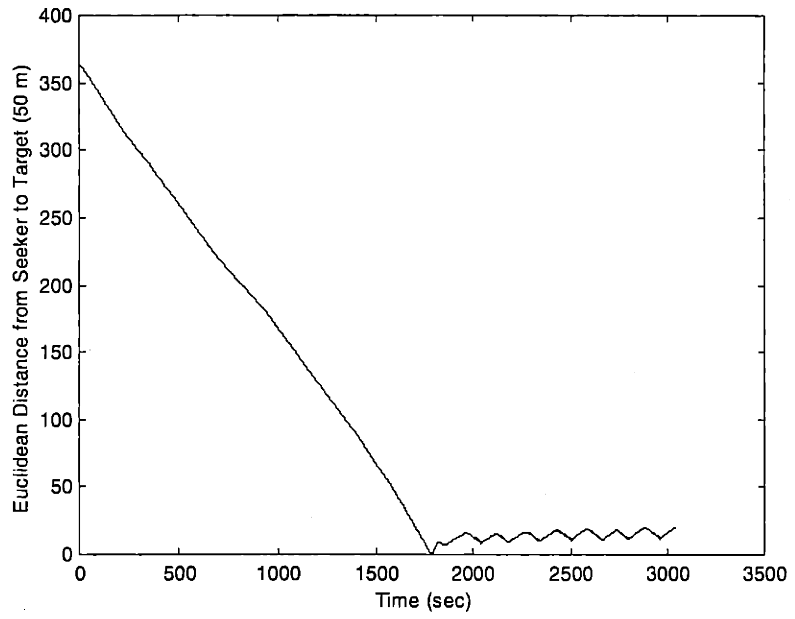


**Figure 5.19 Test Case #7 Error Plot**

## 5.2.8 Test Case #8

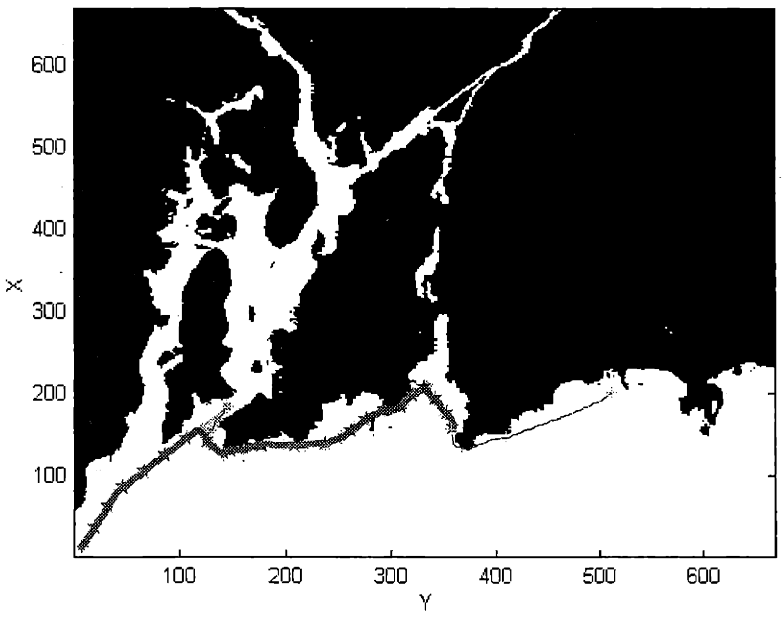


**Figure 5.20 Test Case #8 Track**

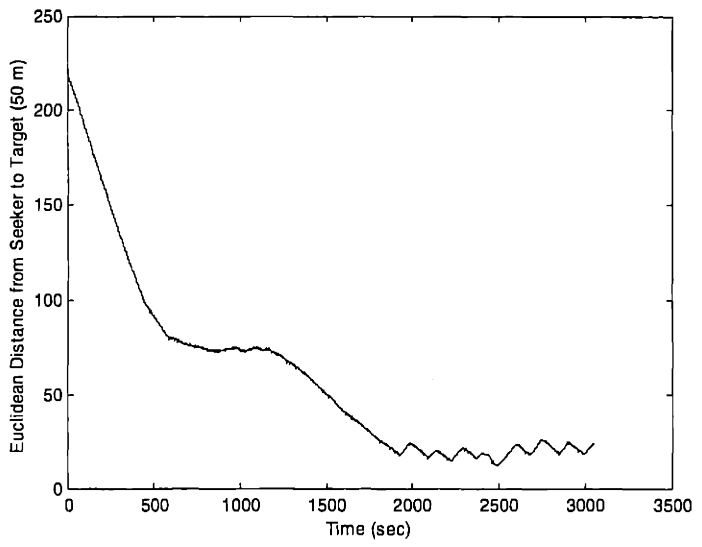


**Figure 5.21 Test Case #8 Error Plot**

## 5.2.9 Test Case #9

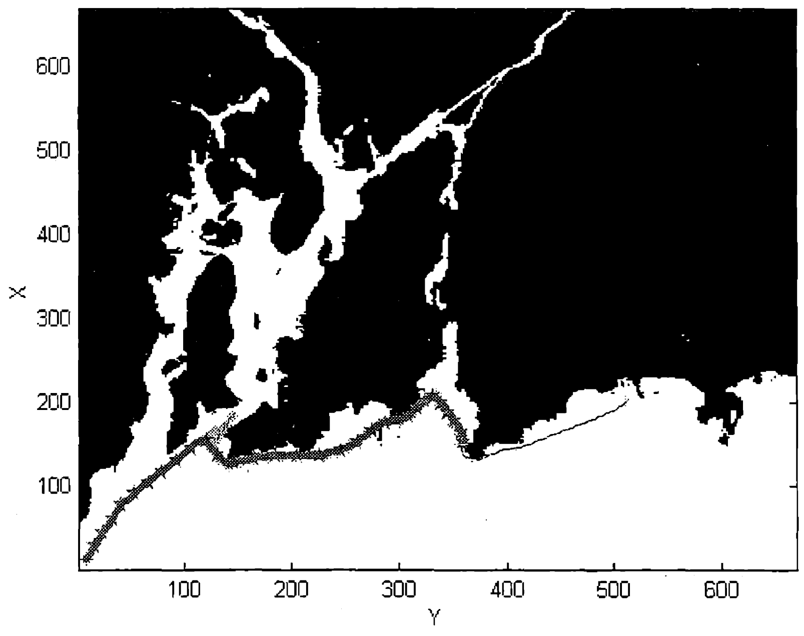


**Figure 5.22 Test Case #9 Track**

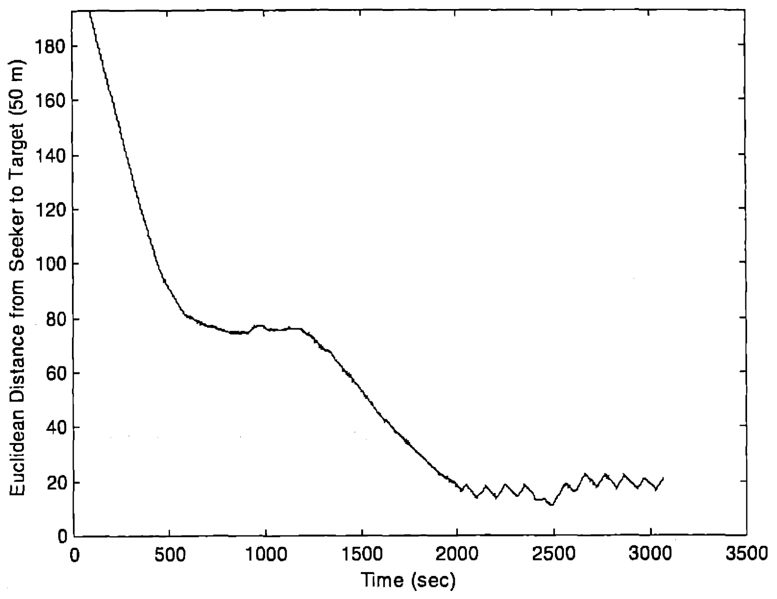


**Figure 5.23 Test Case #9 Error Plot**

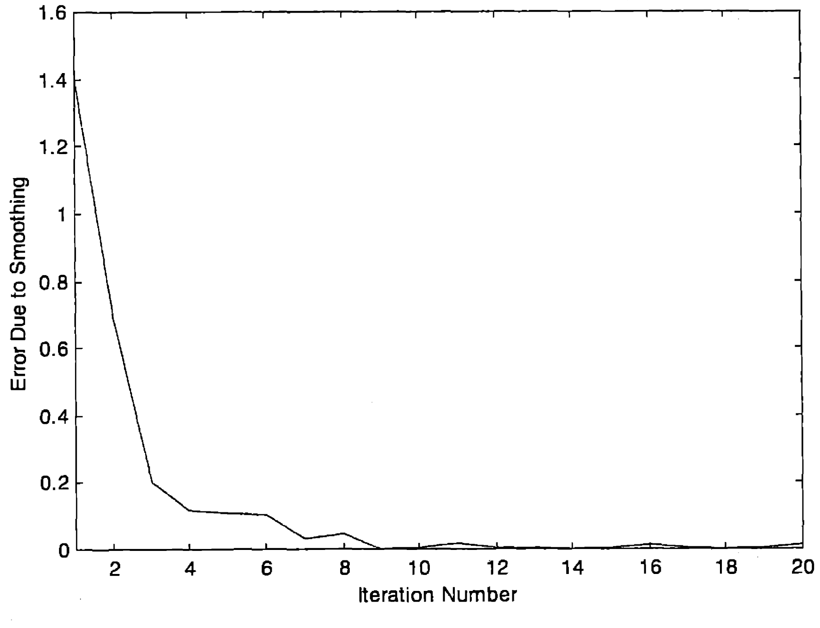


**Figure 5.24 Error due to Smoothing**

Figure 5.24 shows the error due to the smoothing the shortest paths. In the error is displayed for the error due to smoothing twenty of the shortest paths. At every iteration of the path planner, the trajectory must be smoothed. In addition, the maximum computation time over all of these smoothing operations was *0.57* seconds.

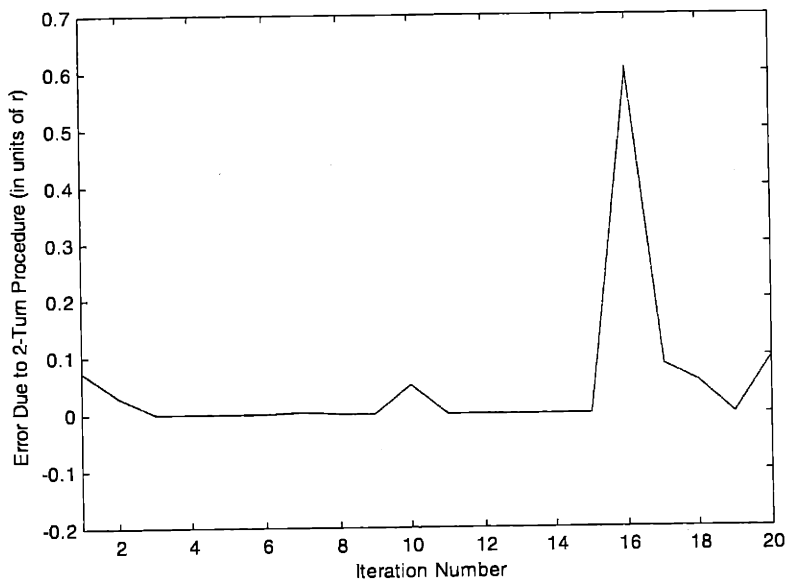


**Figure 5.25 Error Due to 2-Turn Procedure**

Figure 5.25 illustrates the error associated with the 2-turn procedure. Some paths did not require the 2-turn procedure because the initial velocity vectors were already aligned. This plot is in terms of *r*, which was set to *25* meters. This choice of *r* is arbitrary, and is dependent on the vehicle. In context of this application, it was assumed the seeker could accomplish a complete turn inside a grid cell. Thus, the turn radius, *r*, for this vehicle is *25* meters. Therefore, the max error associated with the 2-turn procedure over the entire life of this mission is *15* meters.

## 5.2.10 Test Case #10

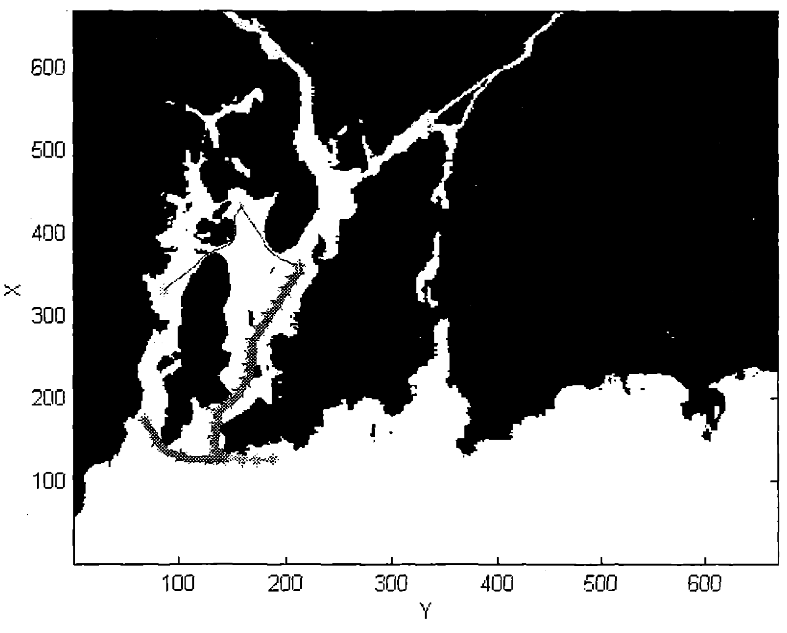


**Figure 5.26 Test Case #10 Track**

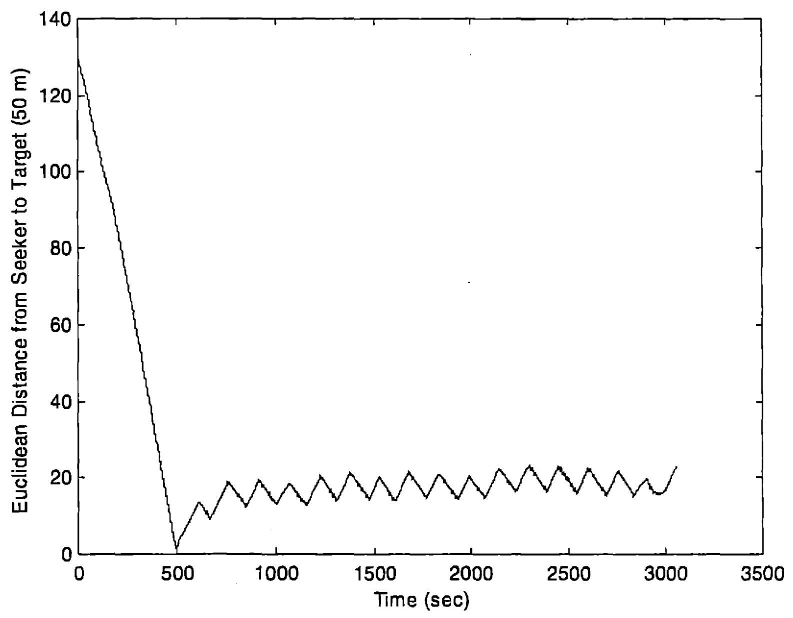


**Figure 5.27 Test Case #11 Error Plot**

## 5.2.11 Test Case #11

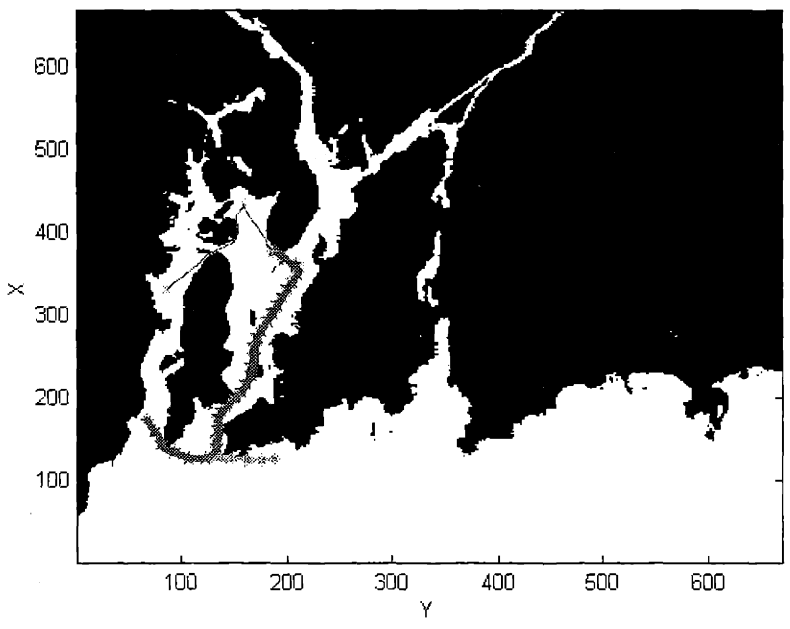


**Figure 5.28 Test Case #11 Track**

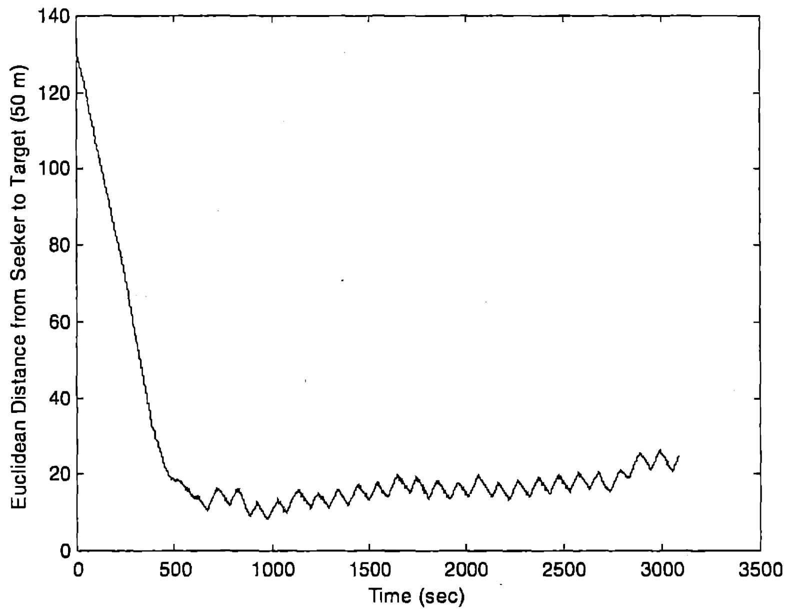


**Figure 5.29 Test Case #11 Error Plot**

### 5.2.12 Test Case #12

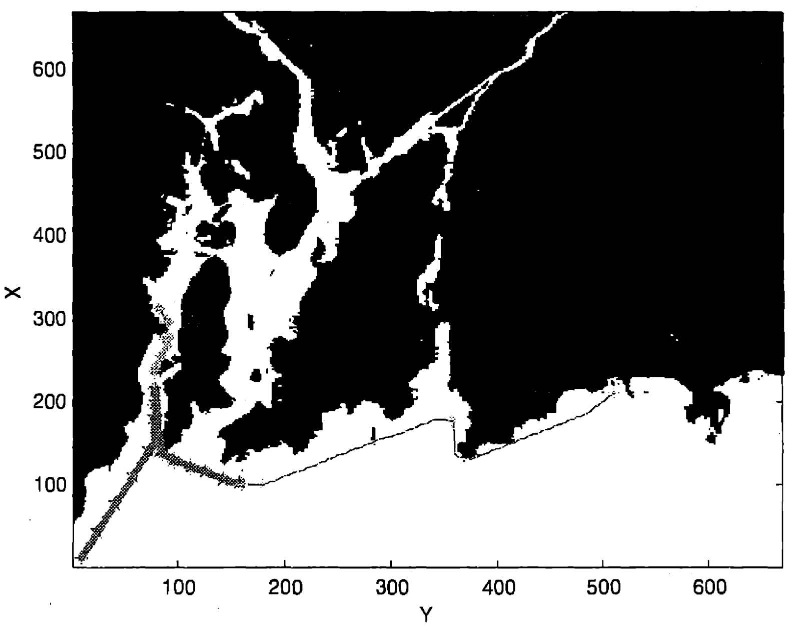


**Figure 5.30 Test Case #12 Track**

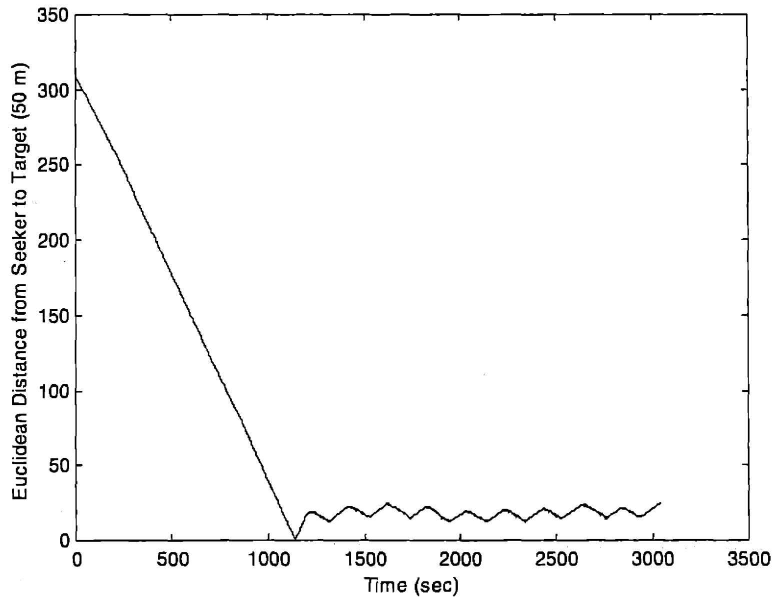


**Figure 5.31 Test Case #12 Error Plot**

## 5.2.13 Test Case #13

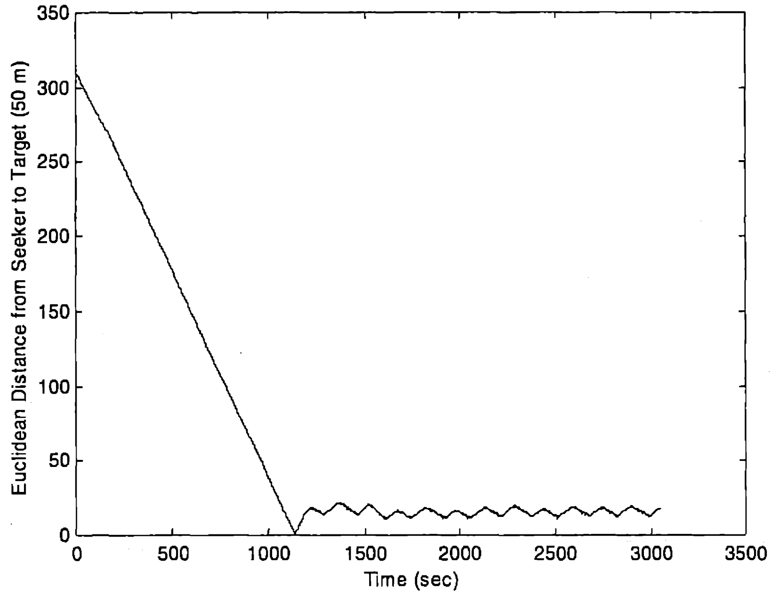


**Figure 5.32 Test Case #13 Error Plot**

## 5.2.14 Test Case #14

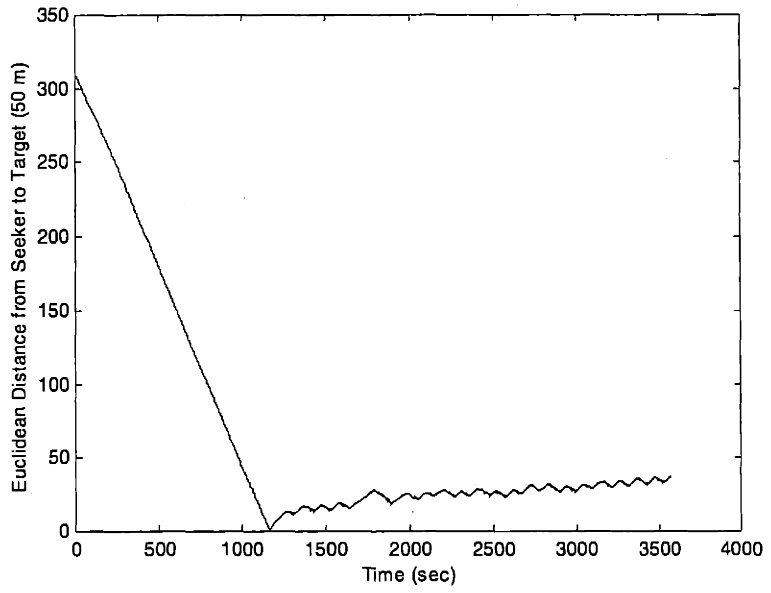


**Figure 5.33 Test Case #14 Error Plot**

## 5.2.15 Test Case #15

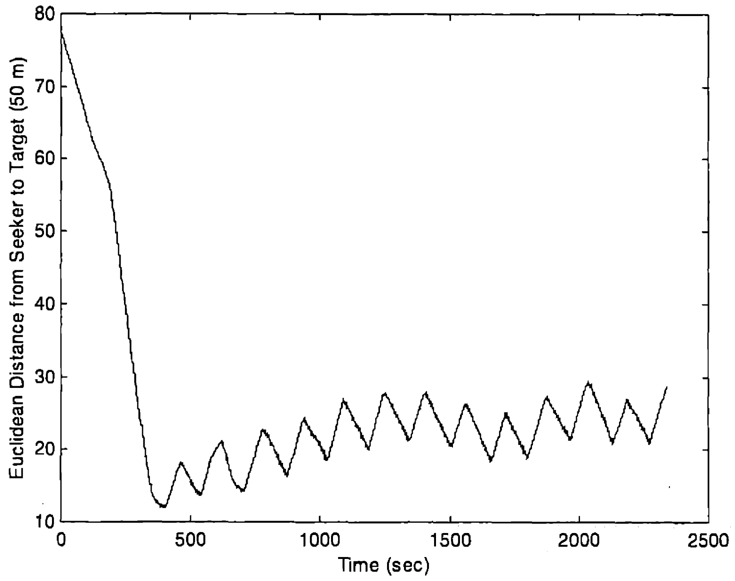


**Figure 5.34 Test Case #15 Error Plot**

## 5.2.16 Test Case #16

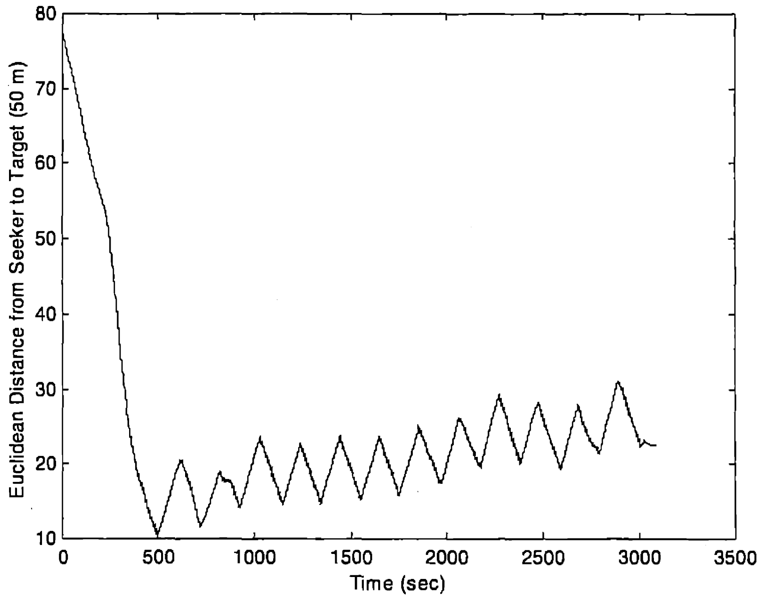


**Figure 5.35 Test Case #16 Error Plot**

### 5.2.17 Test Case #17

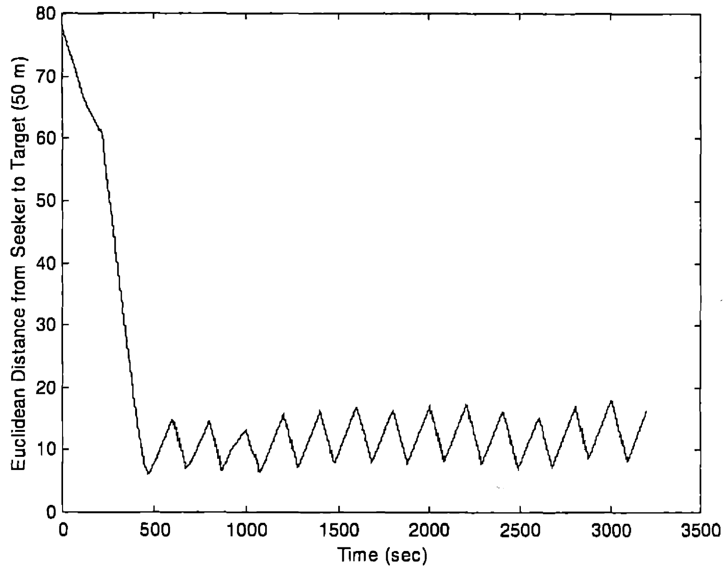


Figure 5.36 Test Case #17 Error Plot

### 5.2.18 Test Case #18

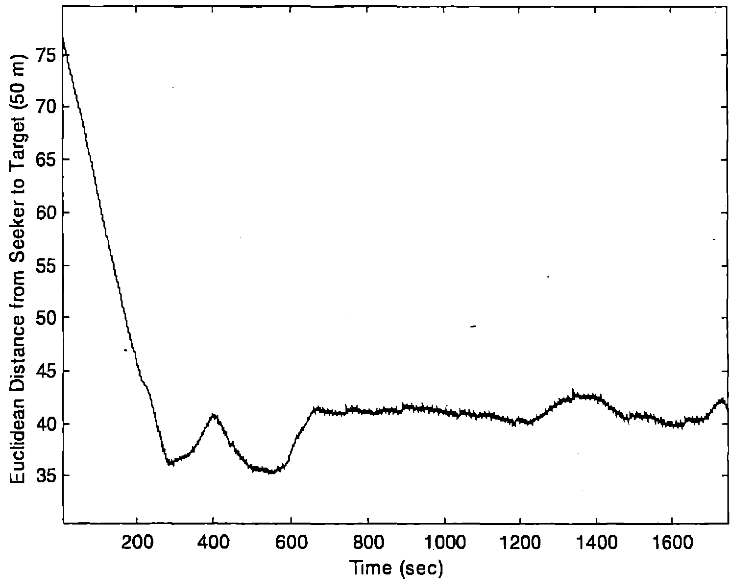


Figure 5.37 Test Case #18 Error Plot

## 5.3 Insight into Test Case Performance

Almost every test case experienced some slow growth in the error bound after the initial distance was decreased. The only two test cases that showed negligible increases in the error were test cases 17, and 18. These two test cases were generated by setting $\Delta c=0$. Thus, the computation time, as conjectured, is expected to have an affect on the distance from seeker to target.

If the seeker is close enough to the target location for the current path, the seeker will reach that target before the next target location is available to the seeker. Therefore, the seeker must wait at the target location of the current path until the next trajectory is available (i.e. the seeker must wait for next the target location to become available and wait for the additional computation time associated with creating a trajectory to that target location). This stop and wait behavior is evident in the end of every test case as demonstrated by the oscillatory nature of the error plots. The following error plot is a magnification of the error plot for test case #16.

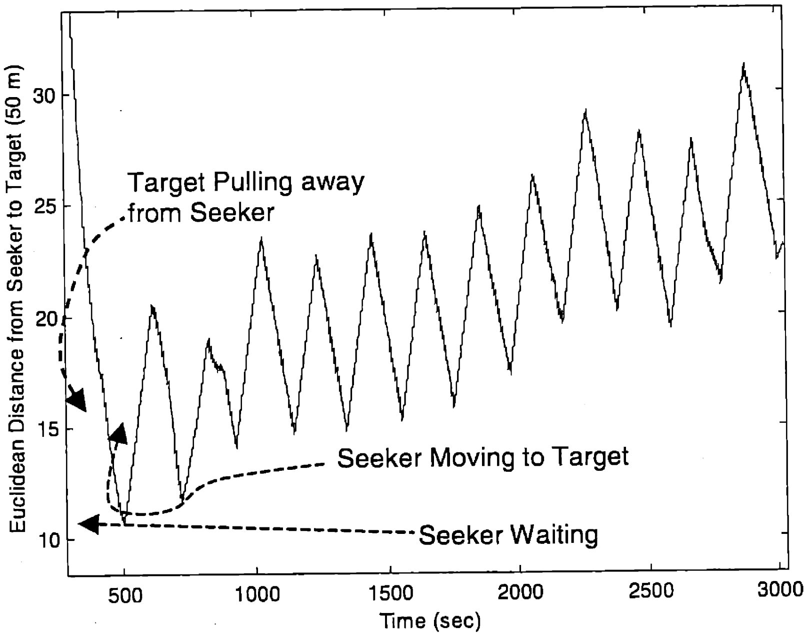


**Figure 5.38 Illustration of Cycling Behavior**

The bottom of each oscillation in the error plot refers to the point at which the seeker first reaches the goal of the current path. At this point, the seeker waits, and the target begins to move away from the seeker. At the peak of each oscillation the seeker begins to move towards the target. Figure 5.38 shows how the maximum distance between the seeker and target begins to slowly increase through time. The peak of each oscillation gives the maximum distance.

The increase in the maximum distance is due to the fixed rate of the target vehicle's position updates. Once the seeker closes the initial distance, the seeker begins to travel between the locations of the target position updates. In the previous chapter, the worst-case scenario was developed as a seeker travelling directly behind a target, travelling a straight line. After the initial distance is closed, the target track figures display a behavior that is not far from the assumption. Therefore, the analysis from the previous chapter should provide insight into the performance of the path planner after the oscillatory behavior in the error appears. The analysis from the previous chapter is repeated and connected to the actual implementation.

The following analysis is for the worst case path, but is provided again for insight into what is happening in the actual test case. In this scenario the seeker is continually reaching the goal of the current path. As defined, the goal locations of each path are the target vehicle's positions as passed to the path planner. Therefore, the distance from one goal to the next goal is $\|V_T\|\Delta t$. The seeker is able to travel for $\frac{\|V_T\|\Delta t}{\|V_s\|}$ units of time and must wait at the current node until a new trajectory is available. Since each iteration of the algorithm lasts $\Delta t+\Delta c$ units of time, the vehicle waits $\Delta t+\Delta c-\frac{\|V_T\|\Delta t}{\|V_s\|}$ at the goal of the current path. This implies a non-zero wait at the goal of the current path. For a given path under this scenario, the target progresses for the entire $\Delta t+\Delta c$ time and the seeker travels for $\frac{\|V_T\|\Delta t}{\|V_s\|}$ time. The target moves $\|V_T\|(\Delta t+\Delta c)$ units, and the seeker moves $\|V_s\|\left(\frac{\|V_T\|\Delta t}{\|V_s\|}\right)$ units towards the target. In order to see the progress made in the direction of the target, the distance traversed by the target is subtracted from the distance traversed by the seeker. This distance is termed $\Delta d_{improvement}$. If $\Delta d_{improvement}$ is positive, improvement is made. If $\Delta d_{improvement}$ is negative then the target moves away from the seeker.

$$\Delta d_{improvement} = \|V_s\| \left( \frac{\|V_T\| \Delta t}{\|V_s\|} \right) - \|V_T\| (\Delta t + \Delta c) \tag{5-1}$$

Simplifying, the above equation yields the following insight.

$$\Delta d_{improvement} = -\|V_T\| \Delta c \tag{5-2}$$

Therefore, under the assumption that the seeker is repeatedly travelling all the way to the current goal, then $\Delta d_{improvement}$ is negative, implying that at each iteration the distance from the seeker vehicle to the target vehicle grows according to (5-2).

*Is this the error seen in practice*? For the test cases chosen, this growth is very close to what is seen. In terms of Figure 5.38 the difference in the peak of one oscillation to the peak of next oscillation refers to $\Delta d_{improvement}$. Clearly, there exist no improvement in the peaks, and from the majority of test cases performed the peaks of these oscillations appear to grow linearly in time. Therefore, a modification of the current scheme is needed.

Once the seeker closes the initial distance and begins to attain the goal of the current path, the improvement at each iteration needs to be non-negative. If $\Delta d_{improvement} \geq 0$, then the peaks of the oscillations in Figure 5.38 will not grow as time progresses. The following section details a few methods of dealing with this condition.

## 5.4 Improvements to Path Planner

### 5.4.1 Wait Approach

Within the context of the fixed sensor updates, the seeker vehicle could decide to wait for an extended time period at a goal node (ignoring any sensor updates), allowing the target vehicle to move sufficiently far away from the goal. Essentially, the seeker is now capable of moving for a further amount of time, because the target vehicle is sufficiently far away. The seeker needs to wait such that the target is at least $\Delta t + \Delta c$ time units away and then use this position as the next goal. The performance expected is similar to the initial closure seen in the error plots. This method albeit acceptable is not developed in context of the simulation.

### 5.4.2 Heuristic, with Fixed $\Delta$t

Up until this point, the path planner uses target locations identified at specific points in time, as the goal to which the seeker is travelling. Theoretical results and simulation, have shown that the vehicle will eventually close the initial distance and then begin reaching the goal of the current path before a $\Delta t$ time segment has expired. Equation (5-2), states that in the worst case the error will increase at each iteration. If simulation results did not support such a conclusion (i.e. the worst case never occurred in practice), then the current methodology may be acceptable. However, this situation is not the case.

*How can the improvement at each iteration be assured of non-negativity?* If we maintain that the target information is incorporated at fixed intervals, a heuristic that guesses where the target may be located in the future may improve the performance of the algorithm. If the seeker can guess the target's location $\varepsilon$ time units before leaving the current goal, it may be possible to reduce the waiting time at the current goal. For instance, if $\Delta t=100$, $\Delta c=3$, and the current path only takes 50 seconds to complete, the seeker must wait for 53 seconds at the current node. The seeker will arrive at the goal of the current path and after 50 seconds of waiting the latest sensor update is provided. Then 3 seconds must expire before a trajectory exists for the seeker. Instead imagine that the seeker is capable of guessing the location of the target after 97 seconds have expired. The seeker is then capable of leaving the current goal along a new trajectory after waiting at the node for only 50 seconds. Now let $\varepsilon$, be defined as the look ahead ability of the seeker. In the example, $\varepsilon = 100-97 = 3$ time units. Since we waited only up to $\Delta t$ - $\varepsilon$ time at the current node, we would expect a performance increase. For simplicity let $\varepsilon = \Delta c$. Now, the vehicle is not penalized for waiting the extra $\Delta c$ time units at a goal. The target now does not travel for the extra $\Delta c$ time period, as present in the previous analysis. Under this scenario what is $\Delta d_{improvement}$?

Under the same assumption, that the vehicle is in the oscillatory behavior (i.e. reaching the goal of the current path), the seeker is able to travel for $\frac{\|V_T\|\Delta t}{\|V_s\|}$ units of time.. Now, the target is travelling, for $\Delta t$ time units. Now, $\Delta d_{improvement}$, becomes

$$\Delta d_{improvement} = \|V_s\|\left(\frac{\|V_T\|(\Delta t)}{\|V_s\|}\right) - \|V_T\|(\Delta t) \tag{5-3}$$

simplifying to,

$$\Delta d_{improvement} = 0 \tag{5-4}$$

The method by which one chooses the estimated position of the target at some time in the future is largely application specific. One method would use the estimated velocity vector of the target and project the current point forward in time. Problems may result if this point is an area that is determined to be an obstacle. At this point one could step in time from *0* to $\varepsilon$, and compute the estimated location. If the estimated position is within an obstacle then use the previous position that is not within an obstacle.

This method was tested on various test cases and demonstrates good performance. In this setting a perfect heuristic is available and is the one used. Later research would need to incorporate a heuristic that models an inexact heuristic. In the following test cases, $\varepsilon = \Delta c$ and $\|V_T\| = .6\|V_S\|$. The first test case demonstrates a typical worst case scenario where the seeker is trying to track a target travelling along a straight line.

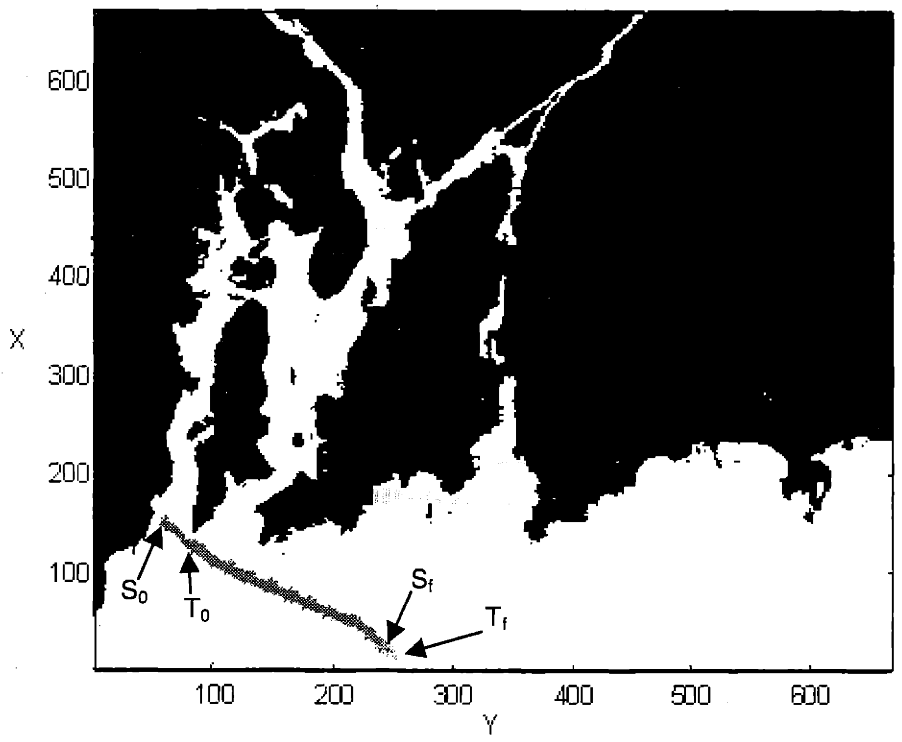


**Figure 5.39 Test Case with Heuristic**

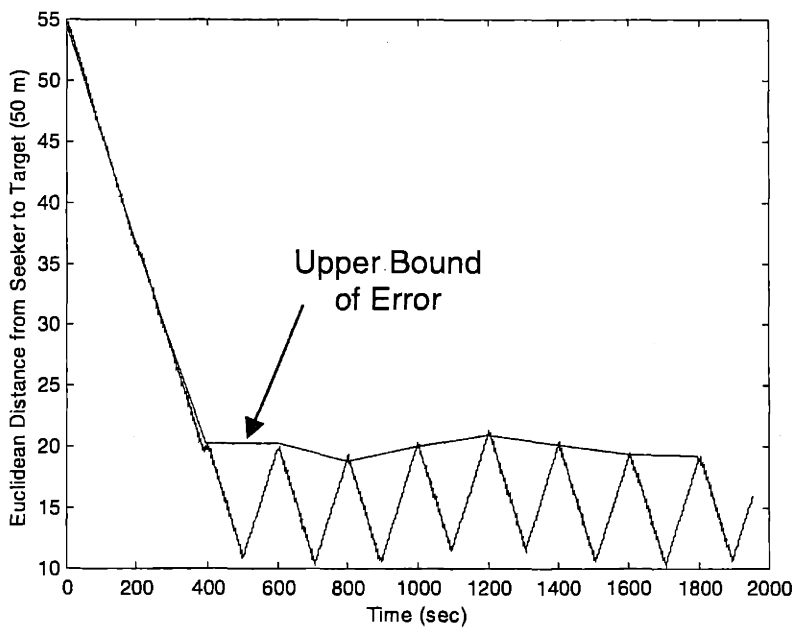


**Figure 5.40 Error with Heuristic**

Figure 5.40 shows how the error does not seem to increase as time progresses. In fact, after the initial distance is closed the improvement at each iteration seems to be close to zero, as expected. A few more test cases were performed to further validate this result.

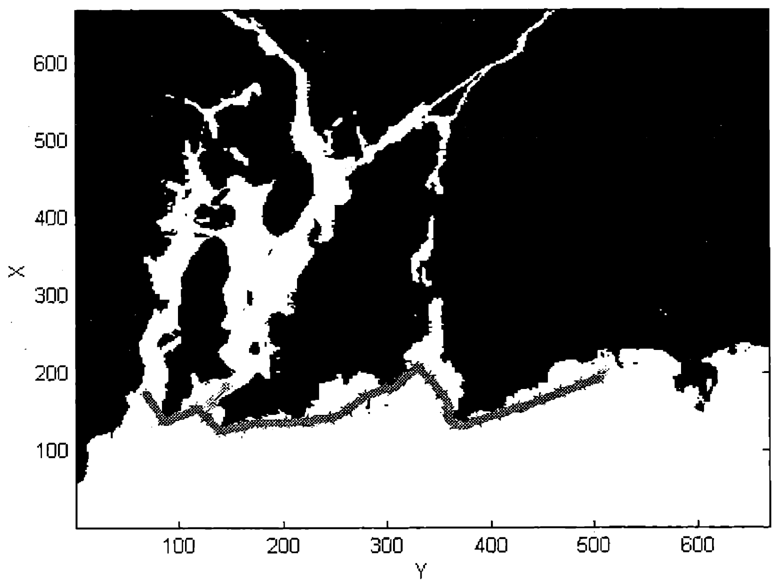


**Figure 5.41 Track #2 Using Heuristic Guess**

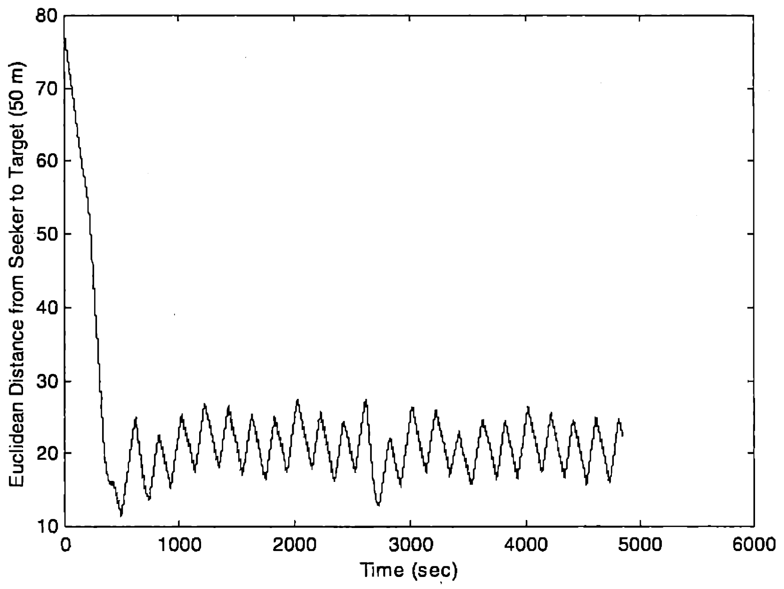


**Figure 5.42 Error Plot #2 with Heuristic Guess**

The second example, is the same track as followed in test case #16. However, now the simulation was allowed to continue for 5000 seconds, lending more credibility to the technique. Once the oscillatory behavior begins, the distance from seeker to target does not seem to grow as time progresses.

## 5.4.3 Variable $\Delta t$ Approach with Varying Seeker Velocity

Another approach to the increasing the error is to incorporate a variable $\Delta t$ into the model. The root of the problem lies in the vehicle having to wait at the goal of the current path. The seeker must wait for a new target location and wait for that target location to be incorporated into a trajectory. This wait is illustrated in Figure 5.38. If this wait were eliminated then growing error could be eliminated. A methodology for performing this approach would dictate that the planner could request the current position of the target, $\Delta c$ time units before the goal of the current path. In this manner, the terminal state condition is eliminated from consideration. Therefore, upon reaching the current node, a trajectory exists in the direction of the target. In this setting, the seeker vehicle does not need to guess where the target vehicle may travel in the future. Immediately upon attaining the goal of the current path, the seeker vehicle begins to traverse a trajectory immediately upon reaching a goal node. However, the seeker is not travelling to the most current location of the target. Rather, the seeker is travelling to the target location found $\Delta c$ time units in the past. If this logic continues the seeker will move closer and closer to the target. However, a condition must be maintained that the length of any path traveled by the seeker must be greater than $//V_S//\Delta c$. Otherwise, the target request that is supposed to occur $\Delta c$ time units before the goal of the current path cannot occur. In order to maintain this condition the speed of the seeker can be adjusted. Specifically, once the seeker vehicle closes to an acceptable distance, the seeker vehicle should attempt to match the speed of the target. This acceptable distance must be constructed such that no path is less than $\Delta c$ in length.

Suppose the seeker is at the acceptable distance threshold, $d_t$, and is travelling at the same speed as the target vehicle. In the worst case path as presented in the previous chapter, the improvement at any iteration becomes zero. This is easy to see because the seeker never has to wait and is travelling for the same amount of time as the target. The lower bound for $d_t$ is $\|V_S\|\Delta c$, if $\|V_T\| = \|V_S\|$. The bound is enforced by the path length criteria (i.e. a path must be greater than or equal to $\|V_S\|\Delta c$). At $d_t = \|V_S\|\Delta c$, the seeker vehicle requests a target update upon leaving the source of the current path. Since the path takes $\Delta c$ time units to complete, a path is ready for travel exactly when the seeker arrives at the goal of the current path. Now the goal of the current path is the source for the next path. Again the seeker requests a new target update after leaving this node, and upon reaching the next goal a path is available. The seeker keeps tracking in this manner, maintaining the $\|V_S\|\Delta c$ separation form the target.

In practice, one may wish to set the lower bound on $d_t$ higher than as described. Since, the computation time is rarely a fixed parameter, one may wish to ensure that the path length is some degree larger than the estimated computation time.

This variable approach given above could be blended with a fixed $\Delta t$ model. When the vehicle is far from the target the fixed $\Delta t$ approach may be more acceptable. For instance, it may benefit the seeker vehicle to update the position of the target while it is travelling along a long trajectory. After the computation is complete the vehicle is then capable of following that new path. Specifically, if a path is longer than $\Delta t_{fixed} + \Delta c$, then at $\Delta t$ on the current path the position of the target is acquired. Thereby, at $\Delta t_{fixed} + \Delta c$, on the current path, the seeker vehicle will begin travelling along the new trajectory. If a path is less than $\Delta t_{fixed} + \Delta c$, then the variable approach should be implemented. In this blended approach, $\Delta t_{fixed}$ is a parameter and set by the mission planner, or set by the system designer.

In implementing this algorithm, some rough adjustments were made to the velocity of the seeker vehicle. If the next path was less than a certain distance then the seeker's speed was set to a fraction of the target's speed. This behavior of traveling slower than the target allows for the distance to grow, and penalizes the seeker for getting to close to the target. If the distance of the next path was within a certain acceptable range then the seeker's speed was set to the speed of the target. If the distance grew larger than a certain multiple of the desired distance then the seeker's velocity was set to the average of the target's speed and the maximum speed of the seeker. Finally, if the distance is over a certain upper threshold level then the seeker is tasked with travelling at its maximum speed. Varying the speeds of the vehicles in this manner is largely an ad hoc approach. More research needs to develop the optimal speed adjustments that should be made.

In order to address the validity of this approach a few simulation runs were performed. It is understood that the manner in which the seeker's velocity vector is chosen is without regard to any optimal choice of the velocity. The method that is employed seems to work in practice and is given below:

```
if $(goal(i+1)_{time} - goal(i)_{time}) < \Delta c$
    $seeker\ speed = .8(target\ speed);$
else if $(goal(i+1)_{time} - goal(i)_{time}) < 2\Delta c$
    $seeker\ speed = target\ speed;$
else if $(goal(i+1)_{time} - goal(i)_{time}) < 4\Delta c$
    $seeker\ speed = (max\ seeker\ speed + target\ speed)/2;$
else
    $seeker\ speed = max\ seeker\ speed;$
```

**Figure 5.43 Rule Used for Adjusting Seeker Speed**

In Figure 5.43, each goal location is time indexed as is meant by the notation $goal(i)_{time}$. The time index refers to the point in time the target location was passed to the path planner. If these times begin to get to close together, then the seeker may get too close to the target location. In addition, if these goals are less than $\Delta c$ apart then a new trajectory may not be available upon the seeker's arrival at the goal of the current path. A sample test run of test case #16 is given to show the performance of this approach. The only change in the test case parameters is that $\Delta t$ is now variable and $\Delta c$ is *15* seconds . The upper bound on $\Delta t$ was set to 200 seconds such that a trajectory lasting longer than $\Delta t + \Delta c = 215$ seconds was not allowed to occur.

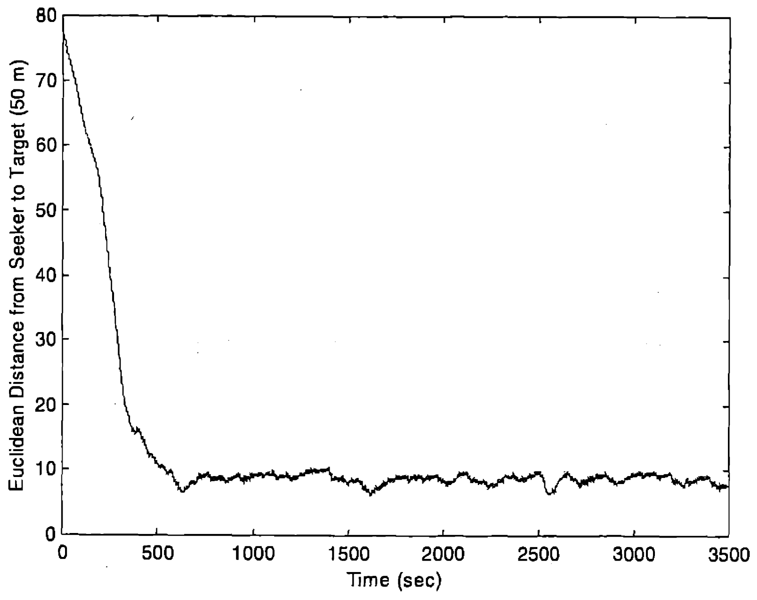


**Figure 5.44 Error Plot with Variable Δt Approach**

Figure 5.44 shows how using the variable approach can maintain about a 10 *x* 50 meter separation from the target for this specific test case. In order for this method to work, the amount of time the seeker spends on each path must be greater than *Δc*. The following plot shows the amount of time the seeker spends on each path.

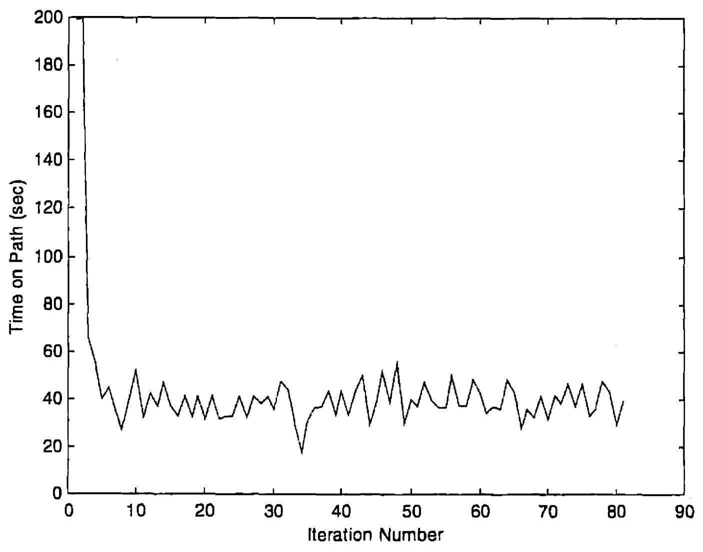


**Figure 5.45 Plot of Path Times**

Figure 5.45 shows a plot of the amount of time the seeker spent on each of the 81 paths created during the simulation. All but one of these paths (iteration #34) is longer in length than the $\Delta c$ lower bound of 20 seconds. Even with the careful adjustments made to the velocity of the seeker, this bound is still capable of being violated. More research into how the speed of the seeker should be adjusted could alleviate this problem. However, within this simulation, the $\Delta c$ value was chosen much larger that the actual time the needed to compute a shortest path. A path was usually ready in under 4 seconds, implying that the bound was not actually violated. If a path is less than the $\Delta c$ specified for the mission, then the seeker cannot get a new trajectory position. Therefore, whenever the seeker leaves the start of the current path, it guesses where the target will be when the seeker reaches the goal of the current path and begins planning for that location. If the seeker makes no new target requests during the current path, this guess will be used for the goal of the next path. The method for the guess is the same as presented in Section 5.4.2. The following trajectory may not be available to the new target location by the time the seeker reaches the goal of the current path. In this case the seeker must remain at the goal of the path until the trajectory information is returned. However, choosing a $\Delta c$ larger than the typical computation time seen in practice, may result in a path being available at the goal of the current path even if the current path took less than $\Delta c$ time units to complete.

In conclusion, the variable $\Delta t$ approach is an improvement over the fixed $\Delta t$ method. With careful adjustments to the seeker's velocity, an upper bound of the distance from the target could be realized. In addition, the stop and wait characteristic seen in the fixed $\Delta t$ method is alleviated in this model. Further research into how one varies the speed of the seeker is needed to fine-tune this approach. In the current implementation the velocity is set for the entire current path. There is no reason that the velocity of the seeker cannot change while it traverses the current path. Such a modification may yield better performance. The variable $\Delta t$ model was run on many of the test cases as presented for the fixed $\Delta t$ model. These results for the subsequent models follow closely to those shown in Figure 5.44 and Figure 5.45 and as such, the plots are not included..

## 5.5 Summary

The framework for the path planner has been developed and implemented within a simulation, showing the effectiveness of the path planner. Even though the fixed *$\Delta t$* case as illustrated in the 18 test cases, shows some limitations, the simple fix using the heuristic approach eliminates the error growth. The results presented within this chapter demonstrate that the analysis for the worst case result is applicable to sample type problems. Therefore, in designing a moving target search one cannot rely solely on target location information that is incorporated at a fixed rate. Even without the heuristic fix, the planner is able to close the initial difference in the two locations. However, once the seeker begins to repeat the terminal state, the growth in the distance from seeker to target begins to grow. This conclusion may seem counterintuitive, however, once the vehicle begins to wait at a node the target still travels and increases its lead. Furthermore, once the terminal states begin to repeat, the distances between goal locations are function of the target's speed and the update rate, *$\Delta t$*. Since, the target never stops moving the distance that the target vehicle improves upon while travelling towards the target is less than the penalty associated with waiting. The penalty is a result of the extra $\Delta$c wait, and by eliminating this wait, the seeker can ensure that the target will not begin to pull away from the seeker.

The variable $\Delta$t approach, is a much better method, and eliminates the terminal state condition. Aside from the advantages this method has in terms of tracking the moving target, from a vehicle controllability point-of-view this type of planner is superior. Making a vehicle wait at a node may not be feasible or it may not be an easy maneuver for the vehicle to handle.

Up to this point, the assumption has been made that *$\Delta c$* is a parameter of the model. Clearly *$\Delta c$* is not a parameter and a real-time variable in the model. The level of performance desired in the tracking capability should be developed for an upper bound of *$\Delta c$,* as found from repeated simulation runs. These simulation runs would have to incorporate the actual hardware on-board the vehicle. If the simulations are of a high enough fidelity, then this determination of *$\Delta c$* should work in practice.

The fixed $\Delta t$ method with or without the heuristic may be simple to implement on certain vehicles. However, in the context of UUVs the variable $\Delta t$ method makes more sense. Tasking a UUV to stop and wait at a certain location may be a very difficult maneuver. For starters, a UUV has momentum and usually sluggish dynamics. Thereby, stopping the vehicle at a certain point is difficult. Secondly, the presence of underwater currents may make hovering at a certain location difficult as well.

The approaches and code developed for this simulation will be the starting point for the track and trail mission envisioned for the Navy's future UUVs. In a few years the Navy hopes to test their next generation UUV test vehicle as it performs the track and trail mission. The work performed in this thesis should prove extremely beneficial to this new test vehicle and the track and trail mission. This test vehicle will then be able to assess the performance of these algorithms on-board an actual UUV.

# Chapter 6

# Conclusions and Future Work

## 6.1 Overview

The objective of this thesis was to create an efficient path planner for a UAV, tasked with tracking a moving target. This problem was tackled as a series of static shortest path problems. A simulation was created that validated the approach taken by the path planner. The simulation provides a tool to analyze the path planner under different scenarios. Given a planar gridded map, the simulation can assess the validity of the moving target path planner for a specific vehicle.

In order to develop the moving target path planner, a thorough description of how the environment is discretized and transformed into a network is given. With the assumption that the vehicle is unable to follow a jagged path, a large portion of this thesis is devoted to methods for smoothing jagged paths. With the network shortest path, and path-smoothing tools in place, the actual path planner is developed. The framework for the path planner is capable of finding continuous paths from a seeker location to a target location as the seeker traverses its environment. Choosing a constant target update rate was shown to demonstrate poor worst case behavior. If the seeker is only able to acquire target information at a fixed rate, then a method employing a heuristic is provided. This heuristic attempts to eliminate this worst case behavior. However, if the vehicle is able to incorporate a variable rate approach then a better method is presented. An added benefit of the variable update approach is the elimination of the terminal state condition.

The methodology and algorithms presented in this thesis will hopefully be put to use an actual vehicle in the near future. The Navy has identified the need for a UUV to perform a track and trail type mission. This thesis provides an efficient approach to this problem and simulation results have demonstrated the feasibility and performance of the path planner on these types of missions.

## 6.2 Planner's Capabilities

The framework for the moving target path planner has been shown to perform well in tracking a moving target. Simulation results have shown that a seeker vehicle is capable of closing the initial distance between the seeker and target, and then maintaining a certain distance from the target throughout the remainder of the mission. The planner exhibits the best performance if variable updates to the target's position are possible. However, if the seeker is limited by fixed rate information then a moving target can still be tracked. However, this approach requires the seeker vehicle to stop and start. Stopping the vehicle and then starting may not be a feasible option on many vehicles. Therefore, the variable rate approach will usually be preferred.

The ability to plan paths that are capable of tracking a moving target can operate independently of the manner in which the actual trajectory is followed. The shortest paths produced are from a graph whose nodes are locations on a planar map of fixed grid size. In order to maintain the slope continuity of a path, the resulting shortest paths are smoothed. The smoothing procedures showed promise in their ability to closely follow typical shortest paths. However, once a path is smoothed the resulting trajectory may cross into an obstacle. The obstacles must then be buffered by an amount equal to the maximum error due to smoothing. The amount of buffering to be performed is largely dependent on the nature of the jagged path that is to be smoothed. For the type of paths seen in the application chosen, a buffering of 2 grid cells should be sufficient. This translates to buffering the obstacles by 100 meters. If the map had higher resolution, the 2 grid buffering result may not be applicable. The shortest paths on these grids may be sufficiently more jagged, causing the smoothing techniques to exhibit greater error. The unavailability of higher resolution bathymetry data limited the ability to perform this analysis. However, as the grid size decreases, it is likely that the amount of buffering (in meters) will also decrease. If the map is stored at 10 meter resolution, a 4 grid buffering may be sufficient (40 meters). It is highly unlikely that with

10 meter grid data, a buffering of 10 grid cells will be needed. Therefore, if the resolution increases the amount of buffering will likely decrease.

In the simulation, the path planner uses a 2-D fixed grid map as the network. Experience gained under the AMMT program suggested that care must be given to the complexity of the shortest path problems. Therefore, this thesis focused on planning without the added dimension of vehicle heading, in hopes of reduces the complexity of the shortest path algorithms. If the vehicle is to travel in the plane, a network whose nodes are locations on the 2-D map may or may not be sufficient. Adding a third dimension, $\theta$, which represents the vehicle's heading may be needed. However, in a gridded map system adding this dimension can cause the network to drastically increase in size. With the map data provided, the ability to plan with only the 2-D grid map is shown to be sufficient in the simulation. If the heading of the vehicle is included as an extra dimension in the network, the path-smoothing component of the planner is no longer needed. Each arc in the path will represent an explicit trajectory along the arc.

The ability of the seeker to track the moving target was measured in the results chapter as the distance between seeker and target through time. In the fixed rate approach the seeker showed good performance at closing any initial distance from seeker to target. However, once this distance is closed, most test cases showed how the distance began to slowly grow through time. Modifying the manner, in which the target location is chosen, allows the fixed rate approach to exhibit better performance. In the presence of a perfect heuristic the error growth is eliminated. The approach using target locations that can be updated at a variable rate, shows more promise of effectively tracking the target. Careful modifications to the seeker's speed can fine-tune this approach so that the seeker maintains some approximate distance from the target. In general, this distance that the seeker maintains between itself and the target is a function of the computation time for producing a shortest path. In the fixed rate approach the distance that is maintained depends more on the rate at which target updates are provided. Since target updates cannot be incorporated at a faster rate than the computation time, the variable approach should exhibit closer tracking. This statement is supported by the simulation results.

Another capability of the path planner is its ability to handle the underwater currents. Since the planning is performed without reference to vehicle heading, the current is not incorporated in the arc costs. Rather, it is used to adjust the heading commands that are generated for a given trajectory. The only restriction on the current in this approach is that the magnitude of the current must be less than the magnitude of the seeker's velocity.

## 6.3 Future Work

The simulation and analysis included in this thesis have demonstrated the feasibility of a moving target path planner. The research performed will serve as a guide for the United States Navy in its future quest to employ unmanned underwater vehicles (UUVs) with the track and trail mission capability. Before these algorithms are put on-board a vehicle some further research may be needed. Likewise, before these algorithms are employed on a vehicle as part of the fleet of the United States Navy some more advanced behaviors may need to be addressed.

### 6.3.1 Network Shortest Paths

The data structure used to store the network was a 2-D array, with the centers of each grid representing a location in the environment. Using a fixed grid size, resulted in much of the memory storing large contiguous regions of *1's* and *0's*. Future research should attempt to incorporate quadtrees or framed quadtrees as a means to store the map and the network. These data structures should decrease the size of the network while maintaining similar properties of the resulting paths. Reducing the size of the network may allow for the vehicle to plan with respect to heading. Incorporating heading into the planning, albeit expensive in terms of network size has many advantages. The advantages included the incorporation of current into path costs and the elimination of path smoothing (and the error inherent in smoothing a path).

Research at Draper Laboratory is currently focusing on the presence of dynamic obstacles, and the ability to repair shortest paths while a vehicle traverses along these paths. The D* algorithm is capable of handling this dynamic nature of the data. The research performed for the moving target planner, assumed the map as static. In an actual implementation this assumption may not prove true. Therefore, research needs to blend the work performed for dynamic obstacles with the moving target planner. Incorporating the changing information into the moving target path planner should not be too difficult as the algorithm used, D*, should be able to repair the paths as needed.

Up until this point no mention has been made of planning in the three spatial dimensions $(x,y,z)$. Currently, plans are either executed at a constant depth or a constant altitude from the sea-floor. Clearly, there may be circumstances where planning in the three spatial dimensions will yield more efficient paths. Nothing in the moving target planner prohibits planning with a 3-D spatial map. In addition, all the smoothing extends to the third dimension. If the extension is made to this dimension, the use of advanced data structures, namely octrees or framed octrees, will be needed. If one then wishes to add vehicle heading, a fourth dimension is needed. Likewise, a fifth dimension, vehicle pitch, may be needed in this type of representation. If all 5 dimensions are used, then a very efficient storage of the map and network is needed.

### 6.3.2 Path Smoothing

As previously mentioned, the path-smoothing was needed due to the choice of not planning with respect to vehicle heading. This choice was motivated by the large network size inherent when vehicle heading is added to the network. If the choice is made to plan in only two dimensions (vehicle position), then smoothing the paths is necessary. As mentioned in Chapter 3, modifications to the 2-turn algorithm could be used to smooth any jagged portion of a path. The advantage to this approach is that the curvature constraint will never be violated. A comparison of the optimization method to the modified 2-turn is an area for future research.

### 6.3.3 Smart Behaviors

If the seeker vehicle is tracking an enemy submarine, then behaviors that eliminate exposure of the UUV to excessive risk of detection will need to be addressed. It may be advantageous to the seeker to travel in a certain relative location to the target vehicle, to minimize detection by the target. In addition, the seeker vehicle may wish to hide along an obstacle while still following its target. A whole host of advanced behaviors may be desired. How these advanced behaviors can be incorporated into the planning of paths is an open area of research.

In addition, the moving target planner that makes use of target locations that are updated at a variable rate, needs further analysis of how the speed of the seeker vehicle is adjusted. Currently, the seeker's speed is set for an entire trajectory depending on how far the current goal is from the seeker's position. Adjusting the speed while traversing along a trajectory may yield a track, which can maintain a constant deviation from the target's location.

## 6.4 Conclusions

The goal of this research, to create an efficient moving target path planner, was attained. Casting the moving target trail mission as a series of shortest path problems in a network allows the problem to be solved in an efficient manner. The resulting path planner is capable of tracking and trailing a moving target. Results from the simulation showed that the worst case behavior for the fixed rate approach is realized in practice. This realization motivated the creation of two modifications to the original approach. Both of these modifications demonstrate the ability of the path planner to overcome the shortcomings of the fixed rate approach.

# Appendix A

## A.1 Description of D* Algorithm

This appendix gives an algorithmic description of the D* algorithm as presented in [15]. The description given here is a modification of Stentz's description. Chapter 2, presented the algorithm as capable of finding the shortest path on a network with limited or dynamic information. Specifically, the algorithm is designed to find shortest paths on a network, where the arc cost change as a vehicle traverses the environment. The changes may result from any general source of dynamic information. The algorithm is designed for a vehicle with an on-board sensor, which senses the environment close to the vehicle. Thus the algorithm is designed to most efficiently handle changes to the environment near to the seeker.

This algorithm is implemented within the path planner simulation. The implementation does not utilize the dynamic nature of the algorithm, as the path-planner assumed static information. Further research into the moving target problem could incorporate changing environment information using the D* algorithm. The advantage, of this algorithm is its explicit storage of the backpointer information. General shortest path algorithms need only store this information for those nodes on the shortest path. This algorithm maintains a complete storage of the backpointer list.

## A.2 Variable Descriptions

The D* algorithm uses the following quantities and structures to operate on a given graph.

- **Open List (S):** This list is used to propagate information about changes in the arc costs. It propagates the information by storing the states whose path estimate has changed but has not been accounted for in the shortest path calculations.

- **t(i):** A tri-state flag associated with every node $i \in N.$ The value of the flag is determined as

$$t(i) = \begin{cases} \text{NEW} & \text{if} \quad i \text{ has never been on the OPEN list} \\ \text{OPEN} & \text{if} \quad i \text{ is currently on the OPEN list} \\ \text{CLOSED} & \text{if} \quad i \text{ is no longer on the OPEN list} \end{cases}$$

  The flag is set to its appropriate value whenever it is inserted or deleted from the OPEN list

- **d(i):** The current estimate of the shortest path from $i$ to destination $t$.
- **o(i):** The optimal shortest path $o(i)$ form $i$ to $t$.
- **b(i):** The backpointer of node $i$. If $i$ succeeds $j$ in the estimated path from $s$ to $t$, then $b(i)=j$.
- **k(i):** The key function $k(i)$, is the minimum of $d(i)$ before modification and all values $d(i)$ assumed by $i$ since it has been placed on the OPEN list.
- **mode(i):** Every node exists in one of two distinct state called *modes*. The mode $i \in N$ is defined as

$$mode(i) = \begin{cases} LOWER \;\; if \;\; k(i) = d(i) \\ RAISE \;\; if \;\; k(i) < d(i) \end{cases}$$

- **$k_{min}$**: $k_{min}$ is the minimum of the key functions of all nodes currently on the OPEN list. $k_{min} = min\{k(i) :\ t(i) = \text{OPEN}\}$.
- **$k_{old}$**: $k_{min}$ prior to the most recent removal of a state from the OPEN list. $K_{old}$ is undefined if no state has been removed from the OPEN list.

The goal of the D* algorithm is to maintain a monotonic sequence $\{X\}$ of states that defines a set of lower bounded path costs for each state $X_i$ that is currently, or was once on the OPEN list. A set of nodes $\{X_1 ... X_n\}$ is called a sequence of states $\{X\}$ if the backpointer of $X_{i+1}$ is $X_i \forall\ i$: $1 \leq i \leq n$. A monotonic sequence, hence, defines a path from source to destination.

Initially all the states are set to NEW, the estimated distance to the source s is set to 0. And the source is placed on the OPEN list (S). After initialization, the algorithm calls the PROCESS-STATE function repeatedly. PROCESS-STATE, when run repeatedly, calculates and returns either the shortest path to the destination t, or determines that the goal can never be reached. After the shortest path has been calculated, the vehicle can start approaching its target on this path. If an obstacle blocks the pre-calculated path, the function MODIFY-COST is called, which corrects the cost of the arcs, and clears the way for the PROCESS-STATE to recalculate the ideal path again.

```
algorithm $D*(G)$
begin
  //initialize variables
  $t(i) := NEW\ \forall i \in N$;
  $d(s) := 0; t(s) := OPEN$;
  $S := \{s\}$;
  while $X \neq t$;
    //compute shortest path
    while $X \in S$
      $PROCESS\text{-}STATE$;
    //shortest path $\{X\}$ is now computed
    proceed on $\{X\}$ until obstacle is detected;
    if obstacle is detected
      $MODIFY-COST$;
  end;
end;
```

PROCESS-STATE starts by removing the node X with the lowest key function k(X) from the OPEN list. If the OPEN list is empty and our current state is not our destination, then our goal is unreachable, and the algorithm should exit and notify user about the infeasibility of the solution.

If X is the RAISE mode, its path might not be optimal. The function first checks if the cost of state X can be lowered. The cost of state X can be lowered, if there is a neighbor of X,Y, that has a d(Y) $+c_{xy}<$d(X). If there is such a neighbor, the backpointer of X is set to Y, and d(X) is set to d(Y) + $c_{xy}$.

If X is in the LOWER mode, then its path cost is optimal, and its neighbors will have to be analyzed. If a neighbor, Y, is found to (1) have a cost greater then the cost of getting to X plus the cost of getting from X to Y, or (2) t(Y) = NEW, or (3) the backpointer of Y is X, but the value of d(Y) is not d(X) + $c_{xy}$, then the backpointer of Y is updated to X, and the cost of Y is set to the cost of X plus the cost of the arc (X,Y). Any neighbors that were altered are entered into the OPEN list.

If X is neither in the LOWER nor RAISE state, then the function ensures that the values of each neighbor is valid and that no looping occurs.

```
procedure $PROCESS - STATE\,()$
begin
  $//check\ for\ feasibility$
  if $|S| < 1$ then $solution\ is\ infeasible;\ exit;$
  $//extract\ state\ X\ that\ has\ k_{min}\ from\ the\ OPEN\ list$
  $X = \{i : k(i) = k_{min}\ and\ i \in S\};\ t(i) = CLOSED;\ k_{old} = k_{min};\ update\ k_{min}$
  if $mode(k_{old}) = RAISE$ then
    if $\left(d(Y) \le k_{old}\ \&\ d(X) > d(y) + c(X,Y)\right)$ then
      $b(X) := Y, d(X) := d(Y) + c(X,Y);$
  if $mode(k_{old}) = LOWER$ then
  $//d(X)\ is\ optimal;\ check\ if\ cost\ of\ neighbors\ can\ be\ lowered$
  for $each\ neighbor\ Y\ of\ X$ do
    if $\begin{pmatrix} t(y) = NEW\ or\ \left(b(Y) = X\ and\ d(Y) \ne d(X) + c(X,Y)\right) \\ or\ \left(b(Y) \ne X\ and\ d(Y) > d(X) + C(X,Y)\right) \end{pmatrix}$ then
      $b(Y) := X;$ insert $Y\ into\ S;\ d(Y) = d(X) + c(X,Y);$
```

```
else
  for each neighbor $Y$ of $X$ do
    //make sure that $d(Y)$ is valid
    if $\left(t(y) = NEW \; or \; \left(b(Y) = X \; and \; d(Y) \neq d(X) + c(X,Y)\right)\right)$ then
      $b(Y) := X$; insert $Y$ into $S$; $d(Y) = d(X) + c(X,Y)$;
    else if $\left(b(Y) \neq X \; and \; h(Y) > h(X) + c(X,Y)\right)$ then
      insert $Y$ into $S$; $d(Y) = d(X) + c(X,Y)$;
    else if $\begin{pmatrix} b(Y) \neq X \; and \; d(Y) > d(Y) + c(X,Y) \; and \\ t(Y) = CLOSED \; and \; d(Y) > k_{old} \end{pmatrix}$ then
      insert $Y$ into $S$;
return;
```

The MODIFY-COST function updates the arc cost function, and places the current state on the OPEN list for reevaluation. If the seeker detects that the cost of reaching a node $Y$ from node $X$ is no the stored value $c(X,Y)$, then MODIFY-COST is called. This causes $X$ to be placed on the OPEN list, S. The changes are propagated through the all descendants of the node $X$. In addition, MODIFY-COST takes the new cost and stores it in c(X,Y).

## A.3 Summary

As implemented within this thesis, the D* algorithm runs like Dijkstra's algorithm. Since the information is static, the MODIFY-COST subroutine is never called into action. Thus, the D* algorithm performs a shortest path search in the same manner as would Dijkstra's algorithm. The information that is used to generate the paths in the moving target planner is the backpointer list. As implemented, the backpointers point to nodes, which are on a path to the current goal. Therefore, if the backpointer list is maintained from one iteration to the next, a path will always exist from the current position to the current goal. This fact is a result of the first iteration of the algorithm that found the shortest path from the seeker's initial position to the target's initial position. The remaining paths are computed from a target location to the next target location. In this manner, if the backpointer list computed in the next algorithm updates the value of the backpointer at a node, the resulting path from that node is optimal. However, for any node that the backpointer is defined the resulting path is not necessarily cost optimal. If the backpointer of node *i*, is not changed in the next iteration of the algorithm, then the resulting path from the node *i* to target exists but is not cost optimal. This path existence argument and the maintenance of the backpointer information are the basis of the moving target path planner.

# Bibliography

[1] Alexander, J. Aircraft Command Interface for a Real-Time Trajectory Planning System. Masters Thesis, MIT, November 1987.

[2] *Bathymetric Data.* U.S. National Geophysical Data Center, Nargansett Bay, Rhode Island. Contributed by University of Rhode Island. (http://www.ngdc.noaa.gov/mgg/gdas/gd_rtv.Html).

[3] Bertsimas, D, and J. Tsitsiklis. *Introduction to Linear Optimization.* Linear Optimization, 1997.

[4] Dijkstra, E. "A Note on Two Problems in Connexion with Graphs." *Numeriche Mathematics* 1, 269-271, 1959.

[5] Dunn, P. "Navy UUV Master Plan". *Proceedings of the International Unmanned Undersea Vehicles Symposium*, Newport, RI, April 2000.

[6] Larson, R. *Calculus: Early Trascendental Functions.* D.C. Heath and Company, Lexington, MA, 1995.

[7] Orlin, J. *Network Flows.* Prentice Hall. Upper Saddle River, New Jersey, 1993.

[8] Osborne, W. LTC, United States Army, et al. *Information Operations, Air Force 2025,* August 1996. (http://www42cs.au.af.mil/au/2025/volume3/chap02/v3c2-1.htm)

[9] Pearl, J. *Heuristics*, Addison-Wesley, 1984.

[10] Ricard, M. "Mission Planning for an Autonomous Undersea Vehicle: Design and Results". *Proceedings of the Technology and the Mine Problem Symposium*, Monterey, CA, November 1996.

[11] Samet, H. *Spatial data structures in Modern Database Systems: The Object Model, Interoperability, and Beyond*, W. Kim, Ed., Addison-Wesley/ACM Press, 1995, 361-385.

[12] Samet, H. "An overview of Quadtrees, Octrees, and Related Hierarchical Data Structures," *in NATO ASI Series, Vol. F40, Theoretical Foundations of Computer Graphics*, Berlin: Spriner-Verlag, 1988.

[13] Simmons, R. Mars Autonomy Project, June 1999. (http://www.frc.ri.cmu.edu/projects/mars/morphin.html).

[14] Stentz, A. "The Focussed D* Algorithm for Real-Time Replanning," *Proceedings International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[15] Stentz, A. "Optimal and Efficient Path Planning for Partially-Known Environments", *Proceedings of the IEEE International Conference on Robotics and Automation*, May, 1994.

[16] Stentz A., Y. Yahja, S. Singh. "Recent Results in Path Planning for Mobile Robots Operating in Vast Outdoor Environments," *Proceedings 1998 Symposium on Image, Speech, Signal Processing and Robotics*, The Chinese University of Hong Kong, September 1998.

[17] Yahja, Y., A. Stentz, S. Singh, and B. Brumitt. "Framed-Quadtree Path Planning for Mobile Robots Operating in Sparse Environments," *Proceedings International Conference on Robotics and Automation*, Leuven Belgium, May 1998.