ABSTRACT

This thesis addresses the problem of guiding an air vehicle through an unknown obstacle laden environment to a specified destination, while maintaining a low altitude. Potential applications for recently developed micro air vehicles (MAVs) are described, illustrating the need for autonomy in such vehicles. A scenario is presented in which an MAV is deployed from a parent vehicle and instructed to fly autonomously to a certain location, without colliding with any obstacles. The MAV must have a guidance algorithm to determine its trajectory, as well as an inner loop controller to actually fly this trajectory. A finite state automaton architecture for the guidance algorithm is proposed. A set of states and logical transitions between them is defined and presented. A series of simulations through several environments is used to test the proposed approach. The results of these tests are presented and analyzed, and the failure modes are explained. The results help to determine the requirements on the MAV's sensor system, and the minimum amount of information and processing required to accomplish the mission. Finally, conclusions are drawn regarding the inherent properties of finite state automata, and future work is recommended to improve upon the automaton approach to obstacle avoidance developed here.

Thesis Supervisor: James D. Paduano
Title: Principal Research Engineer
Table of Contents

1.0 Introduction
1.1 Overview ................................................................. 5
1.2 Potential Applications .................................................. 5
1.3 Related Research ......................................................... 6

2.0 Problem Statement and Preliminary Assumptions
2.1 Problem Statement ....................................................... 8
2.2 Assumptions on Environment .......................................... 8
2.3 Assumptions on MAV ................................................... 8
2.4 Desired Results .......................................................... 9

3.0 Proposed Approach
3.1 Basic Architecture of Guidance Algorithm ......................... 10
3.2 Overview of Logic ....................................................... 10
3.3 State Definitions ........................................................ 13
3.4 State Transitions ........................................................ 14
3.5 Summary of Proposed Approach .................................... 22

4.0 Flight Dynamics
4.1 Aircraft Simulation and Analysis ...................................... 23
4.2 Optimal Velocity ........................................................ 28

5.0 Results
5.1 Test Environments ..................................................... 30
5.2 Failure Modes and Negative Effects ............................... 35
5.3 Robustness ............................................................... 41

6.0 Conclusions and Future Work
6.1 Summary of Architecture ............................................. 42
6.2 Conclusions ............................................................. 42

7.0 References .................................................................. 44

Appendices
Appendix A.1 – MAV Specifications ..................................... 45
Appendix A.2 – Automaton Design Parameters ....................... 46
Appendix A.3 – Simulation Code ........................................... 47
List of Figures

Figure 3.1.1 – System Block Diagram ................................................................. 10
Figure 3.2.1 – Sample Polygon ......................................................................... 11
Figure 3.2.2 – Model of a Building, Segmented into 8 Obstacles ...................... 12
Figure 3.2.3 – Example of Inside Corner .......................................................... 13
Figure 3.3.1 – State Transition Diagram ............................................................ 13
Figure 3.4.1 – Logic flow for state STRAIGHT ............................................. 15
Figure 3.4.2 – Logic flow for state TARGET_HOLD ....................................... 16
Figure 3.4.3 – Logic flow for state MAX_TURN ............................................ 17
Figure 3.4.4 – Logic flow for state HOLD_ALONG_OBS ......................... 18
Figure 3.4.5 – Logic flow for state HEAD_TO_CORNER .............................. 18
Figure 3.4.6 – MAV Trajectory through a Single Building Environment ......... 21
Figure 4.1.1 – Block Diagram of MAV Flight Control System ...................... 23
Figure 4.1.2 – Frequency Response and Root Locus for Flight Path Angle (Gamma).... 24
Figure 4.1.3 – Frequency Response and Root Locus for Altitude ................... 25
Figure 4.1.4 – Frequency Response and Root Locus for Velocity ................... 26
Figure 4.1.5 – Frequency Response and Root Locus for Yaw Rate ................... 27
Figure 4.2.1 – Turning Capabilities of the MAV ............................................. 29
Figure 5.1.1 – Test Environment 1 .................................................................. 30
Figure 5.1.2 – Trajectories through Test Environment 1 ................................. 31
Figure 5.1.3 – Test Environment 2 .................................................................. 32
Figure 5.1.4 – Trajectories through Test Environment 2 ................................. 33
Figures 5.1.5a and 5.1.5b – Sample Trajectories through Harvard Square ....... 34
Figure 5.2.1 – Example of Trajectory Loop from Environment 2 ................... 35
Figure 5.2.2a – Example of Obstacle Collision from Environment 1 ............ 36
Figure 5.2.2b – Close-Up of Obstacle Collision from Environment 1 ............. 37
Figure 5.2.3a – Example of Missed Outside Corner from Environment 1 ...... 39
Figure 5.2.3b – Close-Up of Missed Outside Corner from Environment 1 ...... 40
Figure A.3.1 – Simulink Diagram of MAV Automaton System ..................... 48
Figure A.3.2 – Stateflow Diagram of Guidance Algorithm ............................ 49

List of Tables

Table 3.4.1 – State Identification Numbers ...................................................... 21
Table A.1.1 – MAV Specifications ..................................................................... 45
Table A.2.1 – Automaton Design Parameters .................................................. 46
Table A.3.1 – State Name Map ........................................................................ 47
1.0 Introduction

1.1 Overview

Recent advances in technology have given rise to the development of Micro Air Vehicles, or MAVs. These tiny airplanes, less than 12 inches in wingspan, are being considered for a wide range of applications, in both the military and civilian realms. In all of these applications, control is one of the primary challenges. It is desirable to make the MAV as autonomous as possible, so that it can be given a task and sent on its way, with little to no additional information or processing required from an external source. This paper proposes an automaton approach to this scenario, which will safely guide an MAV through an obstacle field to some given destination.

1.2 Potential Applications

With the development of miniature sensors, many possible uses for MAVs have recently been suggested. Equipped with a camera, MAVs could be used in a combat setting to perform over-the-horizon reconnaissance missions, or photograph enemy territory. They could also perform military reconnaissance in an urban environment. They could be used to deliver sensors to remote locations, detect biological or chemical agents, or locate landmines. In a more civilian context, they could search for survivors after an earthquake or fire. They could even be used to monitor weather or traffic patterns. Due to the limited range of such small aircraft, a multi-vehicle approach is being considered. In this approach, a larger Unmanned Air Vehicle (UAV) carries the MAV to an area near where it must perform its mission. It then deploys the MAV, which can fly at a lower altitude and more easily navigate through an obstacle-laden environment than its parent vehicle. Once the mission is complete, the MAV could either land at some specified destination, or could rendezvous with a larger UAV to return home.

In this context, there are several issues to consider. The parent UAV may have its own mission to carry out, or may be managing multiple MAVs simultaneously. Therefore, it is desirable for each MAV to be able to operate autonomously, and rely on the parent vehicle as little as possible. Ideally, the parent would provide the MAV with basic information about its mission, such as the destination, and then send it on its way. The parent is then free to focus on other tasks, assuming that the MAV’s own controllers will be able to successfully complete the mission. This is analogous to the way an outer control loop treats an inner control loop in a typical feedback control architecture. For example, an outer loop controller may generate a trajectory for a certain vehicle to follow. It inputs this trajectory to the inner loop, which does what it needs to do to make the vehicle follow the assigned trajectory. The outer loop controller is not involved in this process, as long as the inner loop is designed properly. This paper proposes an approach that enables the MAV guidance algorithm to appear to the parent vehicle as simple and independent as an inner loop appears to an outer loop.

A common practice to design a good inner loop controller is to use feedback. Feedback provides stability and minimizes the effects of unwanted disturbances. Optimal system responses can be obtained from a properly designed state feedback controller. In the
above trajectory-following example, the vehicle's position is typically fed back and subtracted from the input (the given trajectory), and then this error is applied to the actuators with some appropriate gain to force the vehicle to the desired position. This practice is both stable and robust, and will successfully drive the vehicle to the desired trajectory. In this thesis, we investigate the hypothesis that there is a method analogous to feedback that can enable an obstacle avoidance algorithm to be stable and to achieve some of the robustness properties of feedback-based control loops.

To test this hypothesis, we propose a finite state automaton architecture for the guidance algorithm. In other words, the algorithm consists of a set of discrete states, such as "straight flight" or "maximum turn." It receives information from the sensors and communication system about the outside world, and combines this with measurements of the vehicle's own position and heading. It then uses a set of logical rules to determine in which state it should be. The state of the guidance algorithm specifies a command or set of commands that are to be issued to the inner loop controller. State transition tests are performed every cycle, using the latest information available. In this way, the algorithm is able to react to its environment and adjust the vehicle's trajectory accordingly, while driving the vehicle to the given destination.

We further postulate that to achieve properties analogous to those of feedback loops, the method should have the following three properties:

- Immediate or nearly immediate use of sensed information about the obstacle field. This could also be termed feedback.
- A simple formulation, based on intuitive notions of the physical relationships governing the obstacle avoidance problem. Simplicity of the method is important for use on an MAV, as the vehicle's small size might greatly limit its onboard memory and processing capabilities.
- Primary reliance on current state, rather than past or predicted state. This implies that the behavior of the vehicle in a particular state will be consistent and straightforward.

This thesis deals primarily with applications in which an MAV must travel through an obstacle-laden environment, such as a city, to reach some specified destination. Therefore, the primary function of the MAV's guidance algorithm is obstacle avoidance. In these applications, we will also assume that the MAV is required to maintain a low altitude, so that simply pulling up and flying over a building is not practical. The obstacle avoidance algorithms employed must be simple, stable, and not heavily dependent on external information. Other researchers have investigated similar obstacle avoidance problems, particularly in the field of robotics. Their findings will now be described.

1.3 Related Research

The problem of obstacle avoidance is not a new one, and many researchers have investigated it and developed their own approaches. For example, Z. Shiller's work deals
with generating the shortest path for a robot to follow through a field of circular obstacles to reach a goal [1,2]. Shiller makes use of the return function, which he defines as the length of the shortest path from any initial point to the goal. To help in evaluating the return function, Shiller also defines “obstacle shadows,” which can be thought of as shadows cast by obstacles when a point light source is located at the goal. If the initial point does not lie in the shadow of any obstacles, then obviously the shortest path is simply a straight line from the point to the goal. If the initial point does lie in the shadow of an obstacle, then the shortest path consists of “a straight line, a constrained arc that follows the obstacle boundary, then a straight line to the goal.” Obstacle-free paths are generated by following the negative gradient of the return function. In environments with multiple obstacles, only the nearest obstacle is processed. Cases in which an obstacle is encountered while another one is being processed are treated separately. Shiller’s results show that by optimally avoiding one obstacle at a time, a near-optimal path through the entire environment is generated. This algorithm works well, and can readily be adapted for our purposes.

H. Choset has also investigated the problem of sensor-based planning [3,4]. In his work, the goal is not to reach some location, but to exhaustively explore some unknown environment. This is done using a type of roadmap called a Voronoi Diagram. This diagram represents the set of all points in an obstacle field equidistant from two or more obstacles. Once a robot has traced out the Voronoi Diagram, then everything about the environment is known. Creating this diagram is not important for our purposes, since the goal here is to reach a destination. However, the ideas encapsulated in the Voronoi Diagram give important information about the environment, as will be demonstrated in section 3.2.

E. Frazzoli has also performed research in the area of motion planning [5,6]. He has developed an algorithm that he terms a maneuver automaton. This automaton uses “maneuvers” to move between “trim trajectories,” thereby creating an obstacle-free path through an environment. This research validates the proposal that continuous systems can be controlled using logic of a discrete nature. However, Frazzoli’s work deals only with vehicles such as helicopters or robots, which can stop anywhere, and turn on a point. Fixed-wing aircraft must remain in motion at all times and have limited turning capabilities. This must be taken into account when developing the guidance algorithm.

Finally, much research has been done in the context of fully known environments, where the goal is to generate an optimal path. The focus of this thesis is unpredictable situations, where the question of optimality becomes irrelevant. It does not make sense to go through laborious computations to determine a trajectory that minimizes some cost, and then quickly discover that that trajectory is no longer feasible, and have to start all over again. It is more important to focus on reaching the destination. By making this the ultimate goal, the resulting trajectory may or may not be optimal, but the mission will be successful.
2.0 Problem Statement and Preliminary Assumptions

2.1 Problem Statement

The goal of this research is to develop an algorithm that will guide an air vehicle from some initial location to some specified destination, while avoiding any obstacles that exist in the environment, and maintaining a low altitude. This algorithm should be simple, requiring as little information and processing as possible. We will use a nonlinear 6 degree-of-freedom simulation of an MAV, adapted from Miotto and Paduano [7], to perform this study. The final result will be a guidance algorithm, to be used as a real-time path planner, driving the MAV to the destination while avoiding obstacles. The secondary goals of this study are to determine the requirements on the vehicle’s sensor system, and to qualitatively investigate the robustness properties of reactive automaton-based algorithms.

No a priori information about the environment is provided to the guidance algorithm. However, some assumptions about both the environment and the MAV’s capabilities must be made. These assumptions will now be described.

2.2 Assumptions on Environment

The environment will be modeled after a city, where the buildings are the obstacles. The MAV will be required to maintain a low altitude, well below the building roofs, but high enough so that trees and utility poles do not pose a danger. The buildings will be polygonal in shape (from an aerial perspective). The MAV will be able to fly in the free space between the buildings. This space would most likely be above the city streets, and so will be scaled accordingly. The destination, or “target” may be slowly moving, but its location is known to the MAV at all times. The target is assumed to be in the same free space as the MAV, with no obstacles within a sufficient radius from it.

In addition, we have decided to treat the problem as a planar one. The requirement of maintaining a low altitude prohibits the MAV from being able to use altitude as a degree of freedom to aid in avoiding obstacles. Therefore, the guidance algorithm will not command altitude changes. It will be the responsibility of the inner loop controller to account for changes in altitude due to turns. All obstacle avoidance and target acquisition logic will be based in a 2-dimensional frame; namely, north and east.

2.3 Assumptions on the MAV

The MAV will know its own location and heading at all times. It will also be equipped with both communication and sensor equipment. The communication system will provide the MAV with target positional data. This information will be updated at a frequency of at least 1 Hz. The MAV will rely on the sensor system for obstacle detection. Further details and requirements on this sensor system are to be determined by the obstacle avoidance logic implementation, and will be described in section 3.2. The MAV will also have adequate computer processing power so that it can react instantaneously to detected obstacles. Although this assumption is unrealistic, actual
delays are expected to be on the order of tens of milliseconds, so that our results will still be applicable. The MAV’s inner loop controller takes the commands generated by the guidance algorithm and generates actuator commands from them, thereby guiding the MAV to the desired trajectory.

2.4 Desired Results

The main result of this research will be the design for a guidance algorithm that will accomplish the goal of guiding an air vehicle through an obstacle field to a specified destination. The requirements for the sensor system on the vehicle will be determined, as well as the minimum amount of information and processing required. A simulation of the proposed architecture will be developed and run, so that the system performance can be analyzed. Finally, some qualitative conclusions will be made regarding the robustness properties of reactive automaton-based algorithms.
3.0 Proposed Approach

3.1 Basic Architecture of Guidance Algorithm

From the problem statement, it can be seen that the MAV must be a reactive system, since it is operating on limited information about its environment. The goal of the system can actually be broken into two separate missions: 1) reach the target; and 2) avoid all obstacles. We propose that the simplest architecture for achieving the missions in this context of limited information is a finite state automaton. This method is consistent with the objective of creating a simple, stable, independent guidance algorithm, and analogies can be made to the feedback approach widely used in inner loop controller design. The following block diagram represents our view of how the guidance algorithm interacts with the outside world and with the aircraft itself.

![System Block Diagram](image)

Figure 3.1.1 – System Block Diagram.

The basic ideas behind the definitions of states and state transitions in the automaton will be described in the following section.

3.2 Overview of Logic

The logic to achieve the goal will make use of the two main references described in section 1.3; namely, Shiller's “obstacle shadows” and Choset's “Voronoi Diagrams.” Shiller states that if the initial point is not in the shadow of any obstacles, then the best trajectory is a straight line from that point to the target. If the initial point does lie in the shadow of some obstacle, then the best trajectory is one which runs parallel to the obstacle boundary until breaking free of the shadow, and then a straight line to the target. Although Shiller's work deals with circular obstacles, the same logic can be applied to line segments.

Because the MAV's sensor system has a finite range, entire buildings will not be detected at once. Rather, small regions will come into view as the MAV approaches them. In order to model this, we divided obstacle boundaries into line segments of various lengths.
A set of these segments forms a continuous, non-overlapping closed contour that represents the outside of a building. In our terminology, the word “obstacle” will refer to one of these individual line segments. Once any portion of an obstacle enters into the sensor system’s range, the obstacle is considered detected. This is consistent with Choset’s idea that “the distance between a point and an obstacle is the shortest distance between the point and all points in the obstacle.” The location of a detected obstacle’s endpoints are known, and the headings of the obstacles on either side of it are also known. The reason this information is important will be described further in a subsequent paragraph.

Clearly, there must be a sensor that can detect obstacles directly in front of the MAV. When such an obstacle is detected, the MAV is commanded to turn to a heading parallel to the obstacle’s heading, and fly in this direction until the obstacle no longer poses a threat. Obviously, there are two such headings parallel to a line segment, which differ by 180°. With limited information, it is impossible to determine which heading is the “best” from the perspective of the ultimate goal of reaching the target. However, collision avoidance is the overriding requirement, even if such avoidance compromises the optimality of the path to the target. Clearly, the MAV should turn to the heading which is closest to its current heading. This prevents the MAV from attempting a very large heading change (or even a circle) that it may not be able to complete without colliding with the obstacle.

We have employed Shiller’s logic to help satisfy the goal of reaching the target. The MAV must have a sensor that can be oriented in the direction of the target at all times. This sensor will detect obstacles that lie in the MAV’s line of sight to the target. In other words, it will detect if the MAV lies in the shadow of an obstacle. If so, then as above, the MAV is commanded to turn to the closest heading parallel to the obstacle’s heading, and fly in this direction until it is no longer in the shadow of this obstacle.

There are several deviations from this basic logic structure, primarily relating to corners of buildings. Corners can be categorized into outside corners and inside corners, according to the following figure:

![Figure 3.2.1 - Sample Polygon.](image)

In this figure, corners 1, 2, 3, 4, and 6 are outside corners, while corner 5 is an inside corner. Both types of corners represent different physical situations, and therefore must
be discriminated, both during sensing and in the guidance algorithm. We have developed a concept which we call "Voronoi vectors" as a convenient way to handle this. Based on the fundamentals of Voronoi Diagrams, a vector can be drawn from every obstacle endpoint in a direction that is equidistant from the two obstacles that meet at that point, and pointed away from the interior of the building. This vector is referred to as the Voronoi vector. The following figure illustrates this concept:

![Model of a Building, Segmented into 8 Obstacles. The corresponding Voronoi vectors are shown.](image)

As mentioned above, when the sensor system detects an obstacle, it must also detect the heading of the two obstacles adjacent to the current obstacle. From this information, the Voronoi vector direction from each endpoint of the current obstacle can be calculated. Therefore, every obstacle has six pieces of information associated with it: the (north,east) coordinates of both endpoints, and the Voronoi vector directions for both endpoints.

Note: In the simulation, the obstacle information is kept in a database. Each obstacle is catalogued separately, so the coordinates and vector associated with each endpoint are actually listed twice, once for each obstacle that emanates from that endpoint.

From the direction of the Voronoi vector, outside corners and inside corners can be identified. If an obstacle in the MAV's line of sight to the target has an outside corner, then it is better to command the MAV towards this corner instead of parallel to the obstacle. This helps drive the MAV around the corner and closer to the target. If an obstacle in the MAV's direct path has an inside corner, special care must be taken to prevent the MAV from having to execute a very sharp turn and/or pass too close to the adjacent obstacle. If the MAV is able to turn parallel to the obstacle in the direction away from the inside corner without requiring a change in heading of more than 135°, then this should be done. Otherwise, the MAV should turn parallel to the obstacle in the direction closest to its current heading; i.e., execute a standard avoidance maneuver. It will then detect the adjacent obstacle and react accordingly. The following figure illustrates this logic.
Figure 3.2.3 – Example of Inside Corner. The heavy lines, labeled A and B, represent 2 obstacles that form an inside corner. The arrows show 2 trajectories approach this corner, both of which will intersect with obstacle B. The inside corner logic dictates that the vehicle following trajectory 1 should turn right, to avoid a collision with obstacle A. However, since trajectory 2 is approaching obstacle B at a much more shallow angle, the logic dictates that the vehicle following trajectory 2 should turn left, and fly parallel to obstacle B towards obstacle A. After making this left turn, obstacle A will be detected, and the vehicle will turn left again.

Only one obstacle at a time is considered, as in Shiller’s work. If more than one obstacle is detected, only the closest one is processed.

3.3 State Definitions

The next figure shows the proposed state transition diagram, which follows from the logic described above. A description of the states follows. The next section will describe the state transitions in detail.

![State Transition Diagram](image)

Figure 3.3.1 – State Transition Diagram.
3.3.1 STRAIGHT

In this state, the MAV is commanded to fly straight along its current heading.

3.3.2 TARGET_HOLD

This state calculates the line of sight from the MAV to the target, then uses proportional feedback to command the MAV to turn to this heading. A limit is imposed on the turning rate to keep the turns gentle.

3.3.3 MAX_TURN

This state takes an obstacle as an input. It commands the MAV to turn parallel to this obstacle, using the maximum turn rate that the MAV can handle.

3.3.4 HOLD_ALONG_OBS

This state also takes an obstacle as an input. It commands the MAV to turn parallel to this obstacle, using proportional feedback with a limit as in the TARGET_HOLD state.

3.3.5 HEAD_TO_CORNER

This state takes an obstacle with an outside corner as an input. It commands the MAV to turn toward a point located a specified distance out from the outside corner, along the Voronoi vector. Proportional feedback with a limit is employed, as in the TARGET_HOLD state. Note that this process requires knowledge of the MAV’s current position.

With these five states, the logic described above can be implemented. The state transitions will now be described in detail.

3.4 State Transitions

The state transitions are the key to encapsulating the proposed obstacle avoidance and target acquisition logic. They are best illustrated using logic flowcharts. The next five figures show the logic flow and subsequent possible transitions from each of the five states. The transitions are then described in detail.
Figure 3.4.1 – Logic flow for state STRAIGHT.
Figure 3.4.2 - Logic flow for state TARGET_HOLD.
Figure 3.4.3 – Logic flow for state MAX_TURN.
Figure 3.4.4 – Logic flow for state HOLD_ALONG_OBS.

Figure 3.4.5 – Logic flow for state HEAD_TO_CORNER.
The individual transitions between states will now be described. For clarity, we will define a LOS obstacle to be an obstacle in the MAV’s line of sight to the target.

3.4.1 STRAIGHT to TARGET_HOLD

If the MAV is traveling straight but not in the direction of the target, and no obstacles are detected from either sensor, then a transition to TARGET_HOLD should be executed to gently turn toward the target.

3.4.2 STRAIGHT to MAX_TURN

If the MAV is flying straight and an obstacle is detected in its direct path, then it should transition to MAX_TURN to avoid this obstacle. Or, if the MAV is flying straight and a LOS obstacle is detected that is not parallel to the current heading of the MAV, then this transition should take place, unless this obstacle lies entirely behind the MAV.

3.4.3 STRAIGHT to HEAD_TO_CORNER

If the MAV is flying parallel to a wall that is in its line of sight to the target, and an outside corner is detected in the direction the MAV is traveling, then a transition should be made to the HEAD_TO_CORNER state to drive the MAV around the outside corner and closer to the target.

3.4.4 TARGET_HOLD to STRAIGHT

This transition is made when the MAV achieves the desired heading toward the target. Or, if in the process of turning toward the target a LOS obstacle is detected that is parallel to the current heading of the MAV, then the MAV should fly STRAIGHT along this obstacle, unless this obstacle has an outside corner in the direction the MAV is traveling.

3.4.5 TARGET_HOLD to MAX_TURN

If the MAV is turning toward the target and an obstacle is detected in its direct path, then it should transition to MAX_TURN to avoid this obstacle. Or, if the MAV is turning toward the target and a LOS obstacle is detected that is not parallel to the current heading of the MAV, then this transition should take place, unless the obstacle lies entirely behind the MAV.

3.4.6 TARGET_HOLD to HEAD_TO_CORNER

If in the process of turning toward the target a LOS obstacle is detected that is parallel to the current heading of the MAV, and this obstacle has an outside corner in the direction the MAV is traveling, then this transition should be made to drive the MAV around the outside corner and closer to the target.
3.4.7 MAX_TURN to HOLD_ALONG_OBS

If the MAV is executing a sharp turn to avoid an obstacle, and the MAV is nearly parallel to this obstacle (within a substantial tolerance), then a transition to HOLD_ALONG_OBS should be made, so that a more gentle turn will be executed.

3.4.8 MAX_TURN to HEAD_TO_CORNER

If the MAV has been directed to execute a sharp turn to avoid a LOS obstacle, and this obstacle has an outside corner in the direction the MAV is headed, this transition should be made to drive the MAV around the outside corner and closer to the target.

3.4.9 HOLD_ALONG_OBS to STRAIGHT

This transition is made when the MAV achieves the desired heading, parallel to the given obstacle.

3.4.10 HOLD_ALONG_OBS to MAX_TURN

If the MAV is gently turning parallel to an obstacle and another obstacle is detected in its direct path, then a transition to MAX_TURN is made to avoid the new obstacle.

3.4.11 HEAD_TO_CORNER to TARGET_HOLD

If the MAV is heading for an outside corner of an obstacle, this transition is made once the obstacle is no longer in the MAV’s line of sight to the target.

3.4.12 Illustrative Example

To better illustrate the state transition logic, here is an example involving a very simple environment.
First, a brief note about the coordinate system: Coordinates are stated as (East, North). Distances are given in meters, and headings are given in degrees. A 0° heading is due north, with clockwise being the positive direction. Consequently, a 90° heading would be due east, and a -135° heading would be southwest.

In Figure 3.4.6, the MAV begins at location (0,0), on a 0° heading. It must reach a stationary target at (0,200), denoted on the plot with a star (*). There is one rectangular building in the environment. For modeling purposes, it has been broken into six segments of 30m in length, labeled A through F. The MAV can detect any of these obstacles within a range of 20m. The numbers along the trajectory indicate state transitions, as listed in the following table:

<table>
<thead>
<tr>
<th>Number</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STRAIGHT</td>
</tr>
<tr>
<td>1</td>
<td>TARGET_HOLD</td>
</tr>
<tr>
<td>2</td>
<td>MAX_TURN</td>
</tr>
<tr>
<td>3</td>
<td>HOLD_ALONG_OBS</td>
</tr>
<tr>
<td>4</td>
<td>HEAD_TO_CORNER</td>
</tr>
</tbody>
</table>

Table 3.4.1 – State Identification Numbers.
The MAV always begins in state STRAIGHT. Since it is initially headed for the target, it remains in this state until an obstacle is encountered. The first obstacle to be detected is obstacle A, that is both in the line of sight to the target, and in the direct path of the MAV. Therefore, a transition to state MAX_TURN is performed, to avoid this obstacle. Obstacle A does not have any outside or inside corners in the direction the MAV is headed (towards obstacle B). Therefore, MAX_TURN will generate the maximum turn rate command that the MAV is capable of handling. The sign of this command, which corresponds to a left or right turn, will be based on whichever direction will orient the MAV parallel to obstacle A with minimal heading change. Here, this means a right-hand turn. Once the MAV is somewhat parallel to obstacle A (within a tolerance of 40°), a transition to state HOLD_ALONG_OBS is made. In this state, gentler, proportional turn rate commands are issued to orient the MAV parallel to obstacle A. Here, this desired heading is 60°. Since no other obstacles are encountered in the meantime, the MAV stays in state HOLD_ALONG_OBS until it is parallel to obstacle A (within a tolerance of 1.7°). Then the MAV transitions to state STRAIGHT, since it has taken the steps necessary to avoid obstacle A.

Now, obstacle A is still in the MAV's line of sight to the target. However, since the MAV is already parallel to it, and there is nothing in its direct path, it remains in state STRAIGHT until it crosses over into the shadow of obstacle B. Since obstacle B has an outside corner in the direction the MAV is headed (towards obstacle C), a transition to state HEAD_TO_CORNER is made. In this state, proportional turn rate commands are issued to orient the MAV on a heading towards an aimpoint. This aimpoint is located at 5m out from the corner formed by obstacles B and C, along the Voronoi vector (the vector which points away from the interior of the building, and bisects the angle formed by obstacles B and C). The MAV will stay in this state and head for the aimpoint until obstacle B is no longer in its line of sight to the target. Once this occurs, a transition to state TARGET_HOLD is performed. In this state, proportional commands are issued to point the MAV towards the target. Since no other obstacles are detected in the meantime, the MAV remains in this state until it reaches the desired heading (within a tolerance of 1.1°). It then transitions to state STRAIGHT and continues flying towards the target. No other obstacles are detected. Some minor corrections are made along the final stretch, and then the MAV successfully reaches the target. This completes the mission.

3.5 Summary of Proposed Approach

The architecture for the MAV's guidance algorithm will be a finite state automaton, with the five states and transitions between them described above. The algorithm will take as inputs the position and heading of the MAV and the location of the target. The MAV will need two sensors, one that will detect obstacles in the MAV's direct path, and one that will detect obstacles in the MAV's line of sight to the target. The guidance algorithm will output a yaw rate command to be issued to the inner loop controller. No other commands are needed. With this architecture, the algorithm will accomplish the mission of guiding the MAV to the target without colliding with any obstacles.
4.0 Flight Dynamics

4.1 Aircraft Simulation and Analysis

The simulation used for testing the proposed obstacle avoidance logic is based upon a nonlinear flight simulator for Matlab and Simulink, developed by Piero Mirotto and James Paduano. A complete description of it can be found in [7]. In this simulation, aerodynamic forces and moments are computed based on calculated aerodynamic derivatives for the MIT MAV, as well as nonlinear aerodynamic effects (i.e., wing stall). This modeling work was performed by Prof. Mark Drela. (See Appendix A.1 for a complete listing of the MAV specifications, including dimensions and stability derivatives.) We made several modifications to this simulation, to allow for stabilization of altitude. A frequency analysis was then performed on the entire aircraft flight control system.

The aircraft model used here is a 3-input, 4-output system, as shown in the following figure. The available inputs are elevator, throttle, and rudder. The system outputs are flight path angle, altitude, velocity, and yaw rate. The system consists of three parts:

- Altitude hold loop, with proportional and derivative (flight path angle) feedback.
- Crude velocity hold loop. This loop was not part of the design work done for this thesis. It is adequate for our needs, and is presented here for completeness.
- Yaw rate command loop, that uses rudder-to-roll attitude coupling to achieve turn coordination. Elevator deflection is instantaneously set to the trim value for a coordinated turn at the commanded yaw rate.

![Block Diagram of MAV Flight Control System](image)

**MAV Augmentation Loops**

Figure 4.1.1 – Block Diagram of MAV Flight Control System.
The following sections describe the open- and closed-loop frequency response and root locus of each channel, with a close-up of the root locus around the origin. The open-loop response is represented with a dotted line, and open-loop poles and zeros are represented by x’s and o’s, respectively. The closed-loop response is represented by a solid line, and the closed loop poles are designated with stars along the root locus.

4.1.1 Flight Path Angle (γ) Feedback

Figure 4.1.2 – Frequency Response and Root Locus for Flight Path Angle (Gamma).

This rate feedback on altitude is similar to pitch angle (θ) feedback, in that it damps the phugoid mode. These poles now lie on the real axis. The other two poles on the real axis represent the degenerate short period mode.

Feeding back the derivative of altitude followed by a proportional feedback on altitude creates a stability augmentation loop. The altitude feedback is described next.
4.1.2 Altitude Feedback

The feedback loop analysis shown here is performed on the system with the flight path angle loop closed at the gain value shown in Figure 4.1.2. Note that the transfer function shown is $(1/s)$ times the elevator-to-flight path angle transfer function.

Since there is no strict requirement on altitude, the gain on altitude feedback is quite low. The time constant is around 25 secs. Good performance was achieved in the simulation runs performed here because there were no steady state disturbances present.

Figure 4.1.3 – Frequency Response and Root Locus for Altitude.
4.1.3 Velocity Feedback

Velocity feedback further damps the phugoid mode. The short period dynamics do not participate here. The open-loop pole located at $-3$ rad/sec is from the motor. This pole couples with the stabilized phugoid mode as gain is increased. From the frequency response plot, it is evident that velocity is not robust to real world steady state disturbances. There is a droop at low frequencies, although the high frequency tracking is good. For our purposes, the poor tracking at low frequencies has no effect, as there are no external disturbances.

Figure 4.1.4 – Frequency Response and Root Locus for Velocity.
4.1.4 Yaw Rate Feedback

The yaw rate feedback loop also acts as a damper for the Dutch roll mode. The rudder is used to induce roll rates. The MAV’s dihedral angles are arranged such that the vehicle enters a banked turn when the rudder is deflected. This is typical of radio-controlled (RC) vehicles with no ailerons. Therefore, yaw rate not only damps the Dutch roll at approximately 30 rad/sec, but it also couples the Dutch roll and roll subsidence poles to create a closed loop dominant mode at 180 rad/sec. A zero at 6 rad/sec causes the severe droop seen in the frequency response curves. The servo poles, located at $-150 \pm 155i$ rad/sec, set the bandwidth of the system. This bandwidth is high enough to allow the system to achieve commanded yaw rates quickly, within about 40 ms.

In the absence of any commands from the guidance algorithm, flight path angle and yaw rate should be driven to zero, while altitude and velocity should take on some steady state value. For altitude, we decided this constant value should be 10m. This altitude is high enough to clear trees and other hard-to-detect obstacles, while low enough to achieve stealth. The gains were chosen such that in all tests that were performed, altitude remained between 8 and 14m. Choosing an optimal value for velocity is a more complicated matter, and will now be described.
4.2 Optimal Velocity

The velocity at which the MAV is flying determines its turning capabilities. By using basic equations for aircraft flight, one can get an idea of the relationship between velocity and maximum turn rate in the nonlinear aircraft we are using here.

From the derivations in [8], three key relationships become evident. The first relates maximum airplane lift to maximum load factor, or \( n_{\text{max}} \).

\[
\frac{1}{2} \rho u^2 S C_{\text{Lmax}} = n_{\text{max}} W
\]

where:
- \( \rho \) = air density
- \( u \) = velocity of aircraft
- \( S \) = wing area
- \( C_{\text{Lmax}} \) = maximum lift coefficient
- \( n_{\text{max}} \) = maximum load factor
- \( W \) = weight of aircraft.

In a coordinated turn, the vehicle’s lift vector is oriented at an angle \( \phi \) from vertical. Therefore, the maintain sufficient lift for level flight, the load factor must satisfy

\[
\cos \phi = 1/n
\]

where:
- \( \phi \) = bank angle.

Finally, during a steady level turn, the vehicle’s centrifugal force must be balanced by a lift component. This leads to:

\[
\psi' = \frac{(g \tan \phi)}{u}
\]

where:
- \( \psi \) = heading of aircraft
- \( \psi' = d\psi/dt \) = turn rate
- \( g \) = gravitational acceleration.

Combining these equations yields:

\[
\psi_{\text{max}}' = \frac{g}{u} * \sqrt{(n_{\text{max}}^2 - 1)}
\]

where

\[
n_{\text{max}} = \frac{1}{2} \rho u^2 S C_{\text{Lmax}} / W
\]
The maximum turn rate is related to the minimum turn radius by the following relation:

$$R_{\text{min}} = \frac{u}{\psi_{\text{max}}}$$

where $R_{\text{min}}$ denotes the minimum turn radius.

The following plot shows the maximum turn rate and minimum turn radius as a function of velocity for the MAV used here.

![Turning Rate vs. Velocity](image1)

![Turning Radius vs. Velocity](image2)

**Figure 4.2.1 – Turning Capabilities of the MAV.** These results come from a coordinated turn analysis of the MAV.

From these plots, one can see that faster velocities yield better turning capabilities, although the turning radius eventually levels off. In practice, we only reliably achieve 2 rad/sec. This may be due to departure during turn entry. We chose to use a velocity of 10m/s, which is well into the flat region of the turning radius curve. This does indeed yield good turning performance.
5.0 Results

5.1 Test Environments

In order to validate the proposed automaton approach and logic algorithms, several test environments were created. These environments were designed to be at the scale of a typical urban city, with the smallest passageways measuring 20m across. This is roughly equivalent to a wide 4 lane road, with parked cars and a sidewalk on each side of the street. As described in section 3.2, the MAV's sensor system will only detect small portions of buildings at a time. These detected line segments will be of various lengths. Also, a series of line segments may not exactly match the true outside edge of a building, due to sensor noise. To model these imperfections, we created the obstacle environments by simply clicking on points along the edge of the building we wished to define, and using a Matlab function to record the coordinates of the points. In this way, long walls are not exactly straight, corners are not perfect right angles, and the obstacles are not of uniform length. This is a good representation of the data that an actual sensor system might send to the algorithm.

The following figure shows the buildings of the first test environment, and how they were modeled. There are 222 individual obstacles, or line segments, with their endpoints denoted by circles. The obstacles range in length from 2.8m to 26.7m, with a mean of 13.5m.

![Test Environment 1](image-url)
This next figure shows the simulated runs through the first test environment. The heavy lines are the building boundaries, and the lighter lines represent the flight trajectories. A total of 64 runs were performed, starting at points every 25 m around the edge of the environment. These locations are denoted by stars (*). The target for each run is the point opposite from the starting location, along a line that passes through the center of the environment. Therefore, each star serves as a beginning point for one run and an ending point for another run, and therefore will have 2 trajectory lines emanating from it. All the targets locations are stationary. Initially, the MAV has a velocity of 10 m/s, and is oriented in a general direction towards the center of the environment. A detection range of 20 m was used.

Out of the 64 runs performed, 62 reached the target, while 2 runs collided with an obstacle. These collisions are denoted by x’s on the figure, around the location of (-55,25). The failed runs originate from (100,200) and (75,200), with respective targets of (-100,-200) and (-75,-200). The failure mode is the same for both runs, and relates to complications arising from situations where the MAV is headed in a general direction away from the target. This will be described in more detail in section 5.2.

The other 62 runs are indeed successful, with most trajectories at least 1m away from any obstacles, and the closest approach being 0.6m. Each run reaches the target within 0.2m. This tolerance is of course related to the angular tolerances used within the logic, and can be adjusted if necessary. The data from these runs was truncated once the MAV reached the target. However, if allowed to run longer, the MAV would naturally remain in the
TARGET_HOLD state and circle the target, if there are no obstacles immediately around it. Other actions could be programmed upon reaching the target, as the particular application demands.

The next two figures show the model of the second test environment, and the trajectories through this environment. These 229 obstacles range in length from 1.9m to 42.4m, with a mean of 15.1m. As before, there are 64 runs, starting at points every 25m around the edge of the environment, with the target at the “opposite” point. The same detection range and initial velocity were employed.

Figure 5.1.3 – Test Environment 2.
In this set, 59 runs reached the target, while 5 runs got stuck in the "courtyard" in the lower left portion of the plot. There were no obstacle collisions. The failure mode for the 5 unsuccessful missions is the same, and is a consequence of working with limited information and no memory of previous locations visited. This will be covered in more detail in a section 5.2. The targets for the 5 unsuccessful runs are \((0,150), (0,125), (0,100), (0,50), \text{ and } (0,25)\), which are located to the left of the courtyard. (The run with target \((0,75)\) follows a different path and does not enter the courtyard at all.)

Of the 59 successful runs, the closest approach to an obstacle is 0.5m, and every trajectory passes within 0.2m of the target. There are 6 runs that enter the courtyard but do not get stuck; the MAV is able to circle and find its way out. The targets for these runs are below the courtyard, at the points \((0,0)\) through \((125,0)\).

In addition to these fabricated environments, it would be useful to see how the system performs in a model of an actual city. To this end, we developed a crude model of the area known as Harvard Square in Cambridge, Massachusetts. This region was modeled with 1283 individual obstacles. These obstacles exhibit quite a range in length, from a minimum of 2m to a maximum of 90m, with a mean of around 27m. An exhaustive set of runs was not performed; rather, individual runs were attempted and analyzed, with a detection range of 15m. These runs exhibited great success, as illustrated by the following figures.
Figures 5.1.5a (top) and 5.1.5b (bottom) – Sample Trajectories through Harvard Square. Each run begins at the open circle, with the destination denoted by a star. No trajectory is closer than 0.75m to any obstacle.
From the simulations runs performed and presented here, it can be seen that the proposed approach to obstacle avoidance is successful in most cases. However, there are a few failure modes that have become evident. These will now be described.

5.2 Failure Modes and Negative Effects

As mentioned above, when working with limited information about the environment, it is often impossible to determine which is the “best” way to turn when an obstacle is detected. We chose to pick the direction that required the least heading change, to prevent the need for excessively sharp turns. Admittedly, this may make the path to the target longer, or may even lead to a dead end. In the case of dead ends, the MAV may be able to turn itself around, if the alley is wide enough. If the alley is too narrow to allow for a “U-turn”, then the MAV will collide with one of the walls. In other cases, the geometry may be such that the MAV gets stuck in a loop, maybe inside a courtyard or around a building. Environment 2 presents such a scenario. The following figure shows one such failed run from environment 2.

![Figure 5.2.1 - Example of Trajectory Loop from Environment 2.](image)

Figure 5.2.1 – Example of Trajectory Loop from Environment 2. The MAV originates from the circle at (450,225). The target is located at (0,125), denoted by the star. The MAV enters the courtyard area as it makes its way toward the target. After detecting the left-hand wall, it turns left, away from the inside corner. Shortly thereafter, the lower wall is detected, which forces the MAV to take another left turn, again away from the inside corner. After becoming parallel to this wall, the logic directs the MAV to turn back towards the target, since there is no need to continue following a wall that is not in the MAV’s line of sight to the target. The trajectory then coincides with the MAV’s initial courtyard entrance, and the same logic sequence repeats. The run was terminated after 150 secs.
There is no way to solve this problem with the information available to the MAV. One possible solution would be to give the MAV some memory capabilities, or some way to recognize when it is stuck in a loop. Then when a loop is detected, alternate logic could be employed to break the cycle.

Another failure mode occurs when the MAV is headed away from the direction of the target. In this case, obstacle shadows are sometimes skewed in an odd way. This creates a situation in which the obstacle that is in the MAV's line of sight to the target and the obstacle with which the MAV should be dealing are different. In most cases, obstacles behind the MAV should be ignored, as they pose no threat. However, if the MAV is flying beside a long wall while heading away from the target, the obstacles which are behind it are indeed important, as they keep the MAV flying parallel to the wall until it passes into the shadow of the next obstacle. The logic attempts to deal with this, by ignoring obstacles whose endpoints lie in the 120° sector behind the MAV and to which the MAV is not already parallel. However, if the MAV is flying parallel to an obstacle that is located behind it, the MAV will stay STRAIGHT until this obstacle is out of range. This is an attempt at resolving the issue, although it is not entirely successful, as the two failures in environment 1 show. The next two figures show the full trajectory of one of those failed runs, and a close-up of the collision area.

Figure 5.2.2a – Example of Obstacle Collision from Environment 1. This figure shows the full trajectory. The MAV originates from (75,200), denoted by a circle, with a target of (-75,-200), denoted by a star.
Figure 5.2.2b – Close-Up of Obstacle Collision from Environment 1. This figure shows the collision location, denoted by an x. In this view, the MAV enters from the right, following the bottom wall. When it detects the wall in its direct path, it executes a right-hand turn, staying away from the inside corner. After becoming parallel to this wall, it attempts to turn back toward the target, since the obstacle in the MAV’s line of sight to the target (the bottom wall) is behind its current location. In the middle of this left-hand turn towards the target, it detects an obstacle in its direct path, and turns to the right to avoid it. Then once again, it attempts a left-hand turn back towards the target as before. Once it detects yet another obstacle in its direct path, it cannot react in time, and collides with the obstacle.

Several logic enhancements were tested to try to eliminate this failure mode. These enhancements involved keeping track of the last obstacle detected, and then not executing any turns in that direction until another obstacle is encountered and processed. In the above example, this would prevent the MAV from turning towards the wall with which it eventually collides. However, in other geometries, this logic leads to further complications, as some obstacles remain in “memory” longer than they should. Since it is difficult to determine when the previous obstacle should be used in logic decisions and when it should be ignored, this logic was removed. Therefore, problems remain in cases where the MAV is headed away from the target, and shadows are skewed in an odd manner.

The effects of this failure mode could be minimized if the MAV were able to react to information instantaneously. MAVs and other types of aircraft have fixed turn radii; they are not able to stop and change direction without changing location. This itself is not a serious problem, it just places a demand on the sensor system to be able to detect
obstacles in a range large enough for the aircraft to react to them. The problem lies in the fact that the turn radius is not fixed, but is dependent upon the internal states of the aircraft. Factors such as velocity and roll angle have a significant impact on the turning capabilities of the aircraft. The time and distance required to execute a particular heading change will be dramatically different, depending on if the aircraft is already turning in that direction, flying straight and level, or turning in the other direction. The HOLD_ALONG_OBS state was created specifically to minimize this effect. By only employing sharp turns for short durations, and then using gentler turns when possible, the chances of having to pull a sharp turn in one direction while in the middle of a sharp turn in the opposite direction are greatly reduced. Also, MAX_TURN only commands a turn rate of 1.25 rad/sec, despite the fact that the MAV is capable of achieving at least a 2 rad/sec turn rate. Even with these modifications, the turn radius and “reaction time” are not constant. In order to account for this, the guidance algorithm would need as inputs some of the internal states of the aircraft. The automaton approach that we are using does not allow for this. Consequently, the MAV may not be able to turn away from a detected obstacle in time, as in the failures of environment 1. With the current approach and limited information available to the guidance algorithm, the best way to prevent this situation from occurring is to keep the MAV from passing too close to any obstacles, so that it can always react in time.

One final failure mode in the simulation relates to the lack of searching for obstacles in the MAV’s direct path while in states MAX_TURN and HEAD_TO_CORNER. This was mainly done as a time-saving measure, with the rationale that while in each of these states, it is highly unlikely that a previously undetected obstacle would “pop up.” However, to make the actual system as reliable as possible, obstacle checking could easily be implemented into these two states.

There are two other potential negative effects of the proposed approach. The first relates to the scale of the environment compared to the thresholds, tolerances, and gains used. (See Appendix A.2 for a complete listing of the automaton design parameters we chose.) It is difficult to develop a set of tolerances and gains that would be appropriate for any scale of obstacles and passageways. An attempt has been made here to vary the lengths of obstacles and widths of streets significantly, to test the robustness of the logic. Overall, the MAV performs well, with just a few minor cases where a different set of tolerances and gains might have been more appropriate. These cases primarily involve outside corners. As described above, when an outside corner is detected, the MAV is instructed to head for an aimpoint, which is placed a specified distance out from the corner. In all runs here, this distance is 5m. In certain geometries, if the MAV passes within the 5m buffer zone, it may start to turn away from the corner to reach the aimpoint, as shown in the following figures.
Figure 5.2.3a – Example of Missed Outside Corner from Environment 1. This figure shows the full trajectory. The MAV originates from (-200,0), denoted by a circle, with a target of (200,0), denoted by a star.
Figure 5.2.3b – Close-Up of Missed Outside Corner from Environment 1. This figure shows a close-up of the outside corner. Because the MAV's starting location is so close to the wall, it remains close to the wall as it flies parallel to it. When it detects the outside corner, it aims for a point 5m out from it. However, since the MAV is flying closer than 5m to the wall, it must turn away from the wall to reach the aimpoint. Therefore, once the MAV is out of the shadow of the obstacle, it is turning left and pointing away from the direction it should be traveling.

While this is not a failure, it is not the intended behavior of the system. In this case, placing the aimpoint closer to the corner would solve the problem, and maintain the intended behavior of driving the MAV around the corner. However, in other cases, this may cause the MAV to pass too close to the corner and possibly collide with the building. Since this is unacceptable, it is better to have a larger buffer zone and take the chance of missing a passageway, instead of having a smaller buffer zone and risking a collision. Out of the 128 runs in the two environments, there were only two missed outside corners.

The other negative effect of the finite state automaton approach is “chatter”, or rapid cycling between two or more states. This occurs when the MAV is hovering around some tolerance which defines a state transition. The tolerances presented here try to minimize this effect.

While there are several failure modes and negative effects of the proposed obstacle avoidance approach, the tests show that the MAV performs very well, with 121 successful runs out of 128, and only 2 serious failures (i.e., collisions with obstacles).
While the testing is not exhaustive, it is extensive enough to give a good measure of the system performance under a variety of conditions.

5.3 Robustness

From the set of simulation runs performed, observations can be made regarding the robustness of both the actual trajectory through the environment, and the avoidance capability of the system. From figures 5.1.2 and 5.1.4, it can easily be seen that the trajectory is very sensitive to the initial position of the MAV. A small change in location can have a great impact on how obstacles are detected, which will in turn completely alter the MAV's path around the obstacles. The trajectory is also sensitive to the design parameters of the system, as described above. Figure 5.2.3 shows a classic example of a design parameter that is not properly "tuned" to the particular situation in which it is being used. If the aimpoint were closer to the corner of the building, then the MAV would round the corner and proceed straight, instead of turning around.

The avoidance capability of the algorithm developed here is indeed robust to changes in initial position, as evidenced by the large number of collision-free simulation runs. However, this capability is also highly sensitive to the system design parameters. Appropriate thresholds, tolerances, and gains are required to ensure the proper behavior of the MAV when an obstacle is detected.
6.0 Conclusions and Future Work

6.1 Summary of Architecture

Through this research, we have developed a guidance algorithm that will accomplish the goal of guiding an air vehicle through an unknown obstacle field to a specified destination. This algorithm also satisfies the requirement of maintaining the vehicle at a low altitude. An effort has been made to keep the algorithms simple, with as little information required as possible.

The guidance algorithm we have developed is a finite state automaton. Its five states involve straight flight, turning towards the target, engaging in a sharp turn to avoid an obstacle, engaging in a more gentle turn to orient parallel to an obstacle, and heading for an obstacle’s outside corner. There are logical transitions between the states, which draw on Shiller’s principle of obstacle shadows and Choset’s concept of Voronoi graphs. The aircraft must be equipped with two sensors, one straight ahead and one that is able to point toward the target at all times. The range of the sensors for a mission in a typical urban environment should be about 20m. The aircraft must also be equipped with instruments that indicate its position and heading at all times. This data, the information from the sensor system, and the target location are all that is required by the guidance algorithm. From some simple calculations and logical decisions, it outputs a single yaw rate command to the aircraft’s inner loop, which will adjust the actuators appropriately, thereby directing the aircraft around obstacles and to the target.

6.2 Conclusions

The simulated runs through the test environments show that the proposed automaton approach to obstacle avoidance is indeed a valid one. However, it has become obvious that the stability of this method is not guaranteed. When working with such limited information, there are situations that cannot be adequately handled. One problem relates to getting stuck in an infinite loop. This could only be resolved by incorporating additional logic to both recognize such a situation and figure out how to break out of it. Another problem relates to the variable reaction time of the aircraft itself. To account for this, the guidance algorithm would have to have access to some of the internal states of the aircraft, primarily velocity and bank angle. These states would be incorporated into the obstacle avoidance logic, so that the aircraft is guaranteed adequate time and distance to react to detected obstacles.

Another conclusion from this research is the fact that there is no universal set of gains and tolerances that can handle every geometrical situation. Therefore, to make an MAV as versatile as possible, the gains, tolerances, and other such design parameters within the logic should be programmable, so that the MAV can function in any scale of environment, and is only limited by its turning capabilities.

Finally, this research has led to an important observation about finite state automata in general. While they appear to be robust, since they are reactive systems by nature, they are actually quite sensitive. In the course of the many simulation runs that we performed,
we found that small changes in the environment or in the MAV’s position and orientation often have a large impact on the system behavior. These small changes can determine which way the MAV turns to avoid an obstacle, which obstacle it processes at a given time, or how sharply and quickly it reacts to situations. In many cases, these small differences determine the success or failure of a mission. The discrete nature of the states, as well as the obstacle “shadow” approach employed here, work well when the system is in the mid-range of such a discrete entity. However, the behavior near the boundaries is highly susceptible to small perturbations. Therefore, applying discrete logic to a continuously changing system must be done carefully, with special attention paid to the boundaries between states.

The next step in the study of finite state automata would be to further analyze system behavior at the state boundaries, and determine if there is a way of making this behavior more robust to small perturbations. Another future research direction pertains to the “stability” properties of discretized dynamical systems. By stability, we mean here that the vehicle avoids collisions. Even if a greatly simplified model of the vehicle were employed, developing guarantees for such a system is a daunting task. However, by limiting the scope of the problem to immediate obstacle avoidance, and separating this from the problem of guaranteeing convergence to an optimal path, progress may be possible.

In terms of this project, future work would entail eliminating the failure modes. In particular, tradeoff studies could be performed to determine how much performance is improved by allowing the guidance algorithm additional information or processing capabilities, or by expanding the sensor suite.

Overall, the research that we performed has shown that an automaton approach can successfully be used to guide an air vehicle through an obstacle laden environment, although additional logic and information is required to guarantee mission completion.
7.0 References


Appendix A.1 – MAV Specifications

The following table shows the relevant parameters of the MAV used in the simulation, including dimensions and stability derivatives.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>50 g</td>
<td>mass</td>
</tr>
<tr>
<td>b_span</td>
<td>28 cm</td>
<td>wingspan</td>
</tr>
<tr>
<td>S_ref</td>
<td>300 cm$^2$</td>
<td>wing reference area</td>
</tr>
<tr>
<td>U_ref</td>
<td>10.0 m/s</td>
<td>CD reference speed</td>
</tr>
<tr>
<td>Alpha_0</td>
<td>0.0349066 rad</td>
<td>baseline alpha_0</td>
</tr>
<tr>
<td>Cm_0</td>
<td>0.019852</td>
<td>baseline Cm_0 at alpha_0</td>
</tr>
<tr>
<td>CL_0</td>
<td>0.266685</td>
<td>baseline CL_0 at alpha_0</td>
</tr>
<tr>
<td>CL_max</td>
<td>2.2</td>
<td>upper stall limit CL</td>
</tr>
<tr>
<td>CL_min</td>
<td>-0.6</td>
<td>lower stall limit CL</td>
</tr>
<tr>
<td>CD_prof</td>
<td>0.05</td>
<td>profile CD</td>
</tr>
<tr>
<td>Uexp_CD</td>
<td>-0.5</td>
<td>CD Re-scaling exponent</td>
</tr>
<tr>
<td>CL_a</td>
<td>4.6199</td>
<td>lift slope</td>
</tr>
<tr>
<td>Cm_a</td>
<td>-0.39336</td>
<td>pitch stability</td>
</tr>
<tr>
<td>CY_b</td>
<td>-0.39662</td>
<td>sideforce due to sideslip</td>
</tr>
<tr>
<td>Cl_b</td>
<td>-0.21680</td>
<td>roll due to sideslip</td>
</tr>
<tr>
<td>Cn_b</td>
<td>0.08010</td>
<td>yaw stability</td>
</tr>
<tr>
<td>CL_q</td>
<td>6.99171</td>
<td>lift due to pitch rate</td>
</tr>
<tr>
<td>Cm_q</td>
<td>-11.66099</td>
<td>pitch damping</td>
</tr>
<tr>
<td>CY_p</td>
<td>-0.33859</td>
<td>sideforce due to roll rate</td>
</tr>
<tr>
<td>CL_p</td>
<td>-0.43663</td>
<td>roll damping</td>
</tr>
<tr>
<td>Cn_p</td>
<td>-0.03182</td>
<td>yaw due to roll rate</td>
</tr>
<tr>
<td>CY_r</td>
<td>0.34889</td>
<td>sideforce due to yaw rate</td>
</tr>
<tr>
<td>Cl_r</td>
<td>0.150439</td>
<td>roll due to yaw rate</td>
</tr>
<tr>
<td>Cn_r</td>
<td>-0.10557</td>
<td>yaw damping</td>
</tr>
<tr>
<td>CL_de</td>
<td>0.016080</td>
<td>lift due to elevator</td>
</tr>
<tr>
<td>Cm_de</td>
<td>-0.03707</td>
<td>pitch due to elevator</td>
</tr>
<tr>
<td>CY_dr</td>
<td>0.00374</td>
<td>sideforce due to rudder</td>
</tr>
<tr>
<td>Cl_dr</td>
<td>0.0</td>
<td>roll due to rudder</td>
</tr>
<tr>
<td>Cn_dr</td>
<td>-0.001670</td>
<td>yaw due to rudder</td>
</tr>
<tr>
<td>CY_da</td>
<td>0.0</td>
<td>sideforce due to aileron</td>
</tr>
<tr>
<td>CL_da</td>
<td>0.0</td>
<td>roll due to aileron</td>
</tr>
<tr>
<td>Cn_da</td>
<td>0.0</td>
<td>yaw due to aileron</td>
</tr>
</tbody>
</table>

Table A.1.1 – MAV Specifications.
Appendix A.2 – Automaton Design Parameters

The following table shows the automaton design parameters used in the simulation, including gains and tolerances.

<table>
<thead>
<tr>
<th>Description of Parameter</th>
<th>Value</th>
<th>States Where Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obstacle detection range</td>
<td>20 m (15 m in Harvard Square runs)</td>
<td>STRAIGHT, TARGET_HOLD, HOLD_ALONG_OBS</td>
</tr>
<tr>
<td>Heading tolerance for determining if MAV is already parallel to obstacle in line of sight to target</td>
<td>0.05 rad</td>
<td>STRAIGHT, TARGET_HOLD</td>
</tr>
<tr>
<td>Heading tolerance for determining if MAV is not pointed toward target</td>
<td>0.03 rad</td>
<td>STRAIGHT</td>
</tr>
<tr>
<td>Heading tolerance for determining if MAV is pointed toward target</td>
<td>0.02 rad</td>
<td>TARGET_HOLD</td>
</tr>
<tr>
<td>Heading tolerance for determining if MAV is nearly parallel to obstacle</td>
<td>0.7 rad</td>
<td>MAX_TURN</td>
</tr>
<tr>
<td>Heading tolerance for determining if MAV is parallel to obstacle</td>
<td>0.03 rad</td>
<td>HOLD_ALONG_OBS</td>
</tr>
<tr>
<td>Angular tolerance for determining if two obstacles have same heading</td>
<td>0.02 rad</td>
<td>HOLD_ALONG_OBS</td>
</tr>
<tr>
<td>Turn rate gain for proportional feedback turns</td>
<td>1.2 rad/sec</td>
<td>TARGET_HOLD, HOLD_ALONG_OBS, HEAD_TO_CORNER</td>
</tr>
<tr>
<td>Turn rate limit for proportional feedback turns</td>
<td>0.6 rad/sec</td>
<td>TARGET_HOLD, HOLD_ALONG_OBS, HEAD_TO_CORNER</td>
</tr>
<tr>
<td>Turn rate for sharp turns</td>
<td>1.25 rad/sec</td>
<td>MAX_TURN</td>
</tr>
<tr>
<td>Specification for identifying inside corners (gives upper bound on angle through the free space between two obstacles that share an endpoint)</td>
<td>2.09 rad. (= 120°)</td>
<td>STRAIGHT, TARGET_HOLD, MAX_TURN</td>
</tr>
<tr>
<td>Specification for identifying outside corners (gives lower bound on angle through the free space between two obstacles that share an endpoint)</td>
<td>4.19 rad. (= 240°)</td>
<td>STRAIGHT, TARGET_HOLD, MAX_TURN</td>
</tr>
<tr>
<td>Distance between outside corner and aimpoint</td>
<td>5 m</td>
<td>HEAD_TO_CORNER</td>
</tr>
</tbody>
</table>

Table A.2.1 – Automaton Design Parameters.
Appendix A.3 – Simulation Code

As mentioned in section 4.1, the flight control portion of the simulation was developed in Matlab Simulink. The guidance algorithm developed here was created in Matlab Stateflow, which is an ideal environment for modeling finite state machines. For intricate computations, Stateflow requires calls to external functions, such as Matlab m-files or C programs. Matlab m-files were used here. Unfortunately, in the current version of Matlab (version 5.3), the interface between Stateflow and the Matlab m-files is quite cumbersome, which imposes limits on the programming style. Presumably, later versions of Matlab will improve upon this interface, to allow for more elegant code.

Note that the names of the states in the simulation differ from those used in this thesis. The following table maps the state names from the names used here to the names used in the simulation.

<table>
<thead>
<tr>
<th>Name of State as Defined in Thesis</th>
<th>Name of State as Defined in Matlab Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRAIGHT</td>
<td>STRAIGHT</td>
</tr>
<tr>
<td>TARGET_HOLD</td>
<td>HEADHOLD</td>
</tr>
<tr>
<td>MAX_TURN</td>
<td>COORTURN</td>
</tr>
<tr>
<td>HOL ALONG_OBS</td>
<td>HEADHOLD_OBS</td>
</tr>
<tr>
<td>HEAD_TO_CORNER</td>
<td>HEADHOLD_CORNER</td>
</tr>
</tbody>
</table>

Table A.3.1 – State Name Map.

The following pages show the Simulink diagram, the Stateflow diagram, and the m-files called from the Stateflow diagram. The m-files are listed as follows:

1. set_environment.m
2. ret_obs.m
3. get_t_ang.m
4. get_obs.m
5. get_dist_array.m
6. dist_to.m
7. get_intersect.m
8. check_ang.m
9. get_hhRc.m
10. get_turn_dir.m
11. get_ctCommand.m
12. get_ctRc.m
13. get_obs2.m
14. get_hhCommand.m
15. get_hhRc_corner.m
16. get_oos.m
Figure A.3.1 – Simulink Diagram of MAV Automaton System.
% SCRIPT: set_environment.m
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01

% NOTES:
% This file is used to define the target and obstacle locations.
% It should be called before the Simulink model is run. All
% coordinates should be given in meters. The variables are:
% Tdata = [Ni El t1; ...
% N2 E2 t2];
% where (Ni,El) is the position of the target at time t1, and (N2,E2)
% is the position of the target at time t2. The simulation linearly
% interpolates between t1 and t2. It does not extrapolate (the
% target is considered to be stationary before t1 and after t2).
% vertices = [Ni1 Ei1 1; ...
% N11 E11 1; ...
% ... ...
% N12 E12 2; ...
% N22 E22 2; ...
% ... ...
% N1n E1n n;
% N2n E2n n;
% ...];
% where (Ni,j,Ei,j) denotes the ith vertex of the jth obstacle. The
% vertices should be listed in order around the obstacle. The code
% will automatically join the first and last points of each obstacle.
% dir = [X1 X2 X3...Xn];
% where Xi is either +1 or -1. Use this variable to specify if the
% vertices are defined in clockwise (-1) or counterclockwise (+1)
% order around the ith obstacle, looking down, with north at 12:00
% and east at 3:00.

Tdata = [200 0 0; ...
  300 0 50];

vertices = [ ...
  40 -10 1; ...
  40 10 1; ...
  60 10 1;
  60 -10 1; ...
  50 50 2;
  50 40 2;
  75 40 2;
  75 50 2;
];

dir = [1 -1]; % counterclockwise = +1, clockwise = -1

N = vertices(end,end);
if length(dir) == N
Obstacles = []; vert = []; 
for i = 1:N, 
    j = 1; 
    for k = 1:size(vertices,1), 
        if vertices(k,3) == i 
            vert(j,:) = vertices(k,:); 
            j = j+1; 
        end; 
    end; 
    Obstacles = [Obstacles; ret_obs(vert, dir(i))]; 
end; 
Ototal = size(Obstacles, 1); 
else 
    disp('*** vertices/dir mismatch ***') 
end;
File 2: ret_obs.m

function [Obstacles] = ret_obs(vertices,dir)

% function [Obstacles] = ret_obs(vertices,dir)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
% INPUTS:
% vertices............matrix of obstacle vertex information. See
% set_environment.m for details.
% dir..................vector specifying in which direction
% obstacles are defined. See
% set_environment.m for details.

% OUTPUTS:
% Obstacles.............matrix of information for all obstacles in
% the environment (endpoint locations and
% Voronoi vector directions)

% NOTES:
% This function is called from set_environment.m, to set up the
% Obstacles matrix from the user inputs vertices and dir. Obstacles
% has the following form:
% Obstacles = [N111 E111 N211 E211 V111 V211; ...
% N121 E121 N221 E221 V121 V221; ...
% ...
% N112 E112 N212 E212 V112 V212; ...
% N122 E122 N222 E222 V122 V222; ...
% ...];
% where (Nijk,Eijk) denotes the ith endpoint (1 or 2) of the jth
% "wall" of the kth "building". Because walls share endpoints,
% (N2jk,E2jk) = (N1(j+1)k,E1(j+1)k).Vijk is the direction of the
% Voronoi vector from the ith endpoint of the jth "wall" of the kth
% "building". This vector points out from the obstacle, at an angle
% which bisects the angle between the 2 walls which meet at that
% endpoint. This function builds up the Obstacles matrix 1 building
% at a time (see set_environment.m for proper calling structure).
% All coordinates are in meters, and vector directions are in radians.

vtotal = size(vertices, 1);
for k = 1:(vtotal-1),
    Obstacles(k,:) = [vertices(k,1) vertices(k,2) ...
        vertices(k+1,1) vertices(k+1,2)];
end;
k = vtotal;
Obstacles(k,:) = [vertices(k,1) vertices(k,2) ...
    vertices(1,1) vertices(1,2)];
Ototal = size(Obstacles,1);

for k = 1:Ototal,
    if k == 12
        end;
        b = k - 1;
    if (b < 1) b = b + Ototal; end;
a = k + 1;
if (a > Ototal) a = a - Ototal; end;
theta1 = atan2(Obstacles(b,4) - Obstacles(b,2), ... 
    Obstacles(b,3) - Obstacles(b,1));
theta2 = atan2(Obstacles(k,2) - Obstacles(k,4), ... 
    Obstacles(k,1) - Obstacles(k,3));
temp1 = (theta1 + theta2)/2;
if temp1 > 0
    temp2 = temp1 - pi;
else
    temp2 = temp1 + pi;
end;
outside = theta1 + dir*pi/2;
if (outside > pi) outside = outside - 2*pi; end;
if (outside < -pi) outside = outside + 2*pi; end;
if (abs(temp1) == pi) temp1 = pi*sign(outside); end;
if (abs(temp2) == pi) temp2 = pi*sign(outside); end;
diff1 = abs(outside - temp1);
diff2 = abs(outside - temp2);
if (diff1 > pi) diff1 = 2*pi - diff1; end;
if (diff2 > pi) diff2 = 2*pi - diff2; end;
if diff1 < diff2
    Obstacles(k,5) = temp1;
else
    Obstacles(k,5) = temp2;
end;
end;
theta1 = atan2(Obstacles(k,4) - Obstacles(k,2), ... 
    Obstacles(k,3) - Obstacles(k,1));
theta2 = atan2(Obstacles(a,2) - Obstacles(a,4), ... 
    Obstacles(a,1) - Obstacles(a,3));
temp1 = (theta1 + theta2)/2;
if temp1 > 0
    temp2 = temp1 - pi;
else
    temp2 = temp1 + pi;
end;
outside = theta1 + dir*pi/2;
if (outside > pi) outside = outside - 2*pi; end;
if (outside < -pi) outside = outside + 2*pi; end;
if (abs(temp1) == pi) temp1 = pi*sign(outside); end;
if (abs(temp2) == pi) temp2 = pi*sign(outside); end;
diff1 = abs(outside - temp1);
diff2 = abs(outside - temp2);
if (diff1 > pi) diff1 = 2*pi - diff1; end;
if (diff2 > pi) diff2 = 2*pi - diff2; end;
if diff1 < diff2
    Obstacles(k,6) = temp1;
else
    Obstacles(k,6) = temp2;
end;
if abs(outside) == pi
    if abs(temp1) > abs(temp2)
        Obstacles(k, 6) = temp1;
    else
        Obstacles(k, 6) = temp2;
    end;
end;
end;
File 3: get_t_ang.m

function [dir] = get_t_ang(Pos1, Pos2, Tdata, tcurr)

% function [dir] = get_t_ang(Pos1, Pos2, Tdata, tcurr)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Pos1....................current North coordinate of MAV (m)
% Pos2....................current East coordinate of MAV (m)
% Tdata...................target data (includes initial and final locations
% and times for the target. See set_environment.m
% for details)
% tcurr...................current simulation time (s)
%
% OUTPUTS:
% dir....................heading from current MAV position to current
% target position (rad)
%
% NOTES:
% For times in between the given initial and final times, the target
% position is linearly interpolated between the given initial and
% final locations. The target is considered to be stationary
% before the initial time and after the final time.

Pos = [Pos1 Pos2];
Tposi = Tdata(1,1:2);
ti = Tdata(1,3);
Tposf = Tdata(2,1:2);
tf = Tdata(2,3);

scale = (tcurr - ti)/(tf - ti);
scale = max([scale 0]);
scale = min([scale 1]);
Tpos = scale*(Tposf - Tposi) + Tposi;

r = Tpos - Pos;
dir = atan2(r(2), r(1));

return;
function [Ocurr] = get_obs(Pos1, Pos2, Obstacles, Thresh, psi, ...
    targ_dir, tcurr)

% function [Ocurr] = get_obs(Pos1, Pos2, Obstacles, Thresh, psi, ...
%    targ_dir, tcurr)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Pos1...............current North coordinate of MAV (m)
% Pos2...............current East coordinate of MAV (m)
% Obstacles.........matrix of information for all obstacles in the
%                  environment (endpoint locations and Voronoi
%                  vector directions)
% Thresh.............threshold distance (m). Only obstacles within
%                    the threshold distance of the MAV will be
%                    considered.
% psi................current heading of the MAV (rad)
% targ_dir...........heading from current MAV position to current
% target
% tcurr..............current simulation time (s)
%
% OUTPUTS:
% Ocurr...............index of current obstacle to be avoided. See
% NOTES for details.
%
% NOTES:
% This function looks for obstacles in the line of sight from the MAV
% to the target, and in the MAV's direct path, within the threshold
% distance. If an obstacle is found that must be avoided, Ocurr is
% set to the index of that obstacle. The global variables LOS and
% HIT are true/false flags that denote if the current obstacle is in
% the line of sight or the direct path, respectively. If the MAV is
% already parallel to the obstacle deemed "in the way", then Ocurr is
% set to -1 and the MAV will fly STRAIGHT. If a parallel obstacle is
% in the MAV's line of sight to the target, and has an outside corner
% in front of the MAV, then Ocurr is set equal to the index of this
% obstacle plus 10000. This is interpreted as a flag to transition
% to the HEADHOLD_CORNER state.

Ocurr = 0;
Pos = [Pos1 Pos2];
global LOS HIT;
LOS = 0;
HIT = 0;

dist_array = get_dist_array(Obstacles, Pos, targ_dir, Thresh);

Ocurr = find(dist_array == min(dist_array));
Ocurr = Ocurr(1);
if min(dist_array) == 1e10
    Ocurr = 0;
end;
if Ocurr ~= 0  % check if obstacle in LOS is also in direct path
    obs = Obstacles(Ocurr,:);
    tol = 1e-12;
    [xint, yint] = get_intersect (obs, Pos, psi);
    ang = atan2((yint - Pos(2)), (xint - Pos(1)));
    rel_ang = abs(psi - ang);
    if (rel_ang > pi) rel_ang = 2*pi - rel_ang; end;
    if (rel_ang < pi/2)
        ((((obs(3) - xint) > tol) & ((xint - obs(1)) > tol)) | ...
         (((obs(1) - xint) > tol) & ((xint - obs(3)) > tol))) | ...
        (((obs(4) - yint) > tol) & ((yint - obs(2)) > tol)) | ...
        (((obs(2) - yint) > tol) & ((yint - obs(4)) > tol)))
        HIT = 1;
        LOS = 1;
    end;
end;
if Ocurr ~= 0 & HIT ~= 1
    LOS = 1;
    obs = Obstacles(Ocurr,:);
    theta2 = atan2(obs(4) - obs(2), obs(3) - obs(1));
    theta1 = atan2(obs(2) - obs(4), obs(1) - obs(3));
    if 1 - abs(cos(theta2 - psi)) <= 0.00125  % parallel, w/tolerance
        % of 0.05 rad
        dist_array = get_dist_array(Obstacles, Pos, psi, Thresh);
        Ocurr_new = find(dist_array == min(dist_array));
        Ocurr_new = Ocurr_new(1);
        if min(dist_array) == le10  % no obstacles in direct path
            HIT = 0;
            if ((1 - cos(theta2 - psi) <= 0.00125) & ...
                (cos(theta2 - obs(6)) < -0.5))
                [xint, yint] = get_intersect ([obs(3) obs(4) ... obs(3)+5*cos(obs(6)) obs(4)+5*sin(obs(6))], Pos, psi);
                ang = atan2((yint - Pos(2)), (xint - Pos(1)));
                rel_ang = abs(psi - ang);
                if (rel_ang > pi) rel_ang = 2*pi - rel_ang; end;
                if rel_ang < pi/2
                    Ocurr = Ocurr + 10000; % good outside corner (>240 deg.)
                else
                    Ocurr = -1;  % outside corner is behind MAV
                end;
            elseif ((1 - cos(theta2 - psi) <= 0.00125) & ...
                (cos(theta2 - obs(5)) < -0.5))
                [xint, yint] = get_intersect ([obs(1) obs(2) ... obs(1)+5*cos(obs(5)) obs(2)+5*sin(obs(5))], Pos, psi);
                ang = atan2((yint - Pos(2)), (xint - Pos(1)));
                rel_ang = abs(psi - ang);
                if (rel_ang > pi) rel_ang = 2*pi - rel_ang; end;
                if rel_ang < pi/2
                    Ocurr = Ocurr + 10000; % good outside corner (>240 deg.)
                else
                    Ocurr = -1;  % outside corner is behind MAV
                end;
            else
                Ocurr = -1;  % no good outside corner
            end;
        else
            Ocurr = -1;  % no good outside corner
        end;
    else
        Ocurr = 0;  % obstacle in LOS is also in direct path
    end;
end;
else
    Ocurr = Ocurr_new;
    HIT = 1;
    LOS = 0;
end;
else
    HIT = 0;
    ang1 = atan2((obs(2) - Pos(2)), (obs(1) - Pos(1)));
    rel_ang1 = abs(psi - ang1);
    if (rel_ang1 > pi) rel_ang1 = 2*pi - rel_ang1; end;
    ang2 = atan2((obs(4) - Pos(2)), (obs(3) - Pos(1)));
    rel_ang2 = abs(psi - ang2);
    if (rel_ang2 > pi) rel_ang2 = 2*pi - rel_ang2; end;
    if (rel_ang1 > 2*pi/3) & (rel_ang2 > 2*pi/3)
        Ocurr = 0; % if both endpoints of obstacle in LOS are
        % behind MAV, ignore obstacle
        LOS = 0;
    end;
elseif HIT == 1 % no obstacles in LOS to target, check immediate path
    LOS = 0;
    HIT = 0;
    diff = abs(psi - targ_dir);
    if (diff > pi) diff = 2*pi - diff; end;
    if abs(diff) > 0.52 % only check immediate path if psi differs
        % from LOS by >30 degrees (time-saving measure)
        dist_array = get_dist_array(Obstacles, Pos, psi, Thresh*3/4);
        Ocurr = find(dist_array == min(dist_array));
        Ocurr = Ocurr(1);
        if min(dist_array) == 1e10 % no obstacles in direct path
            HIT = 0;
            Ocurr = 0;
        else
            HIT = 1;
        end;
    end;
end;
return;
File 5: get_dist_array.m

function [dist_array] = get_dist_array(Obstacles,Pos,traj,Thresh)

% function [dist_array] = get_dist_array(Obstacles,Pos,traj,Thresh)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Obstacles..............matrix of information for all obstacles in
% the environment (endpoint locations and
% Voronoi vector directions)
% Pos.....................vector of current coordinates (North, East)
% of MAV (m)
% traj.....................heading of line through Pos, along which
% obstacles will be looked for (rad)
% Thresh..................threshold distance (m). Only obstacles
% within the threshold distance of the MAV
% will be considered.
%
% OUTPUTS:
% dist_array..............array of obstacle distances (m). See NOTES
% for details.
%
% NOTES:
% This function first finds all obstacles within the threshold
% distance of the MAV position. It then determines if a ray starting
% from Pos with a heading of traj will intersect with any of these
% obstacles. If so, then the value of dist_array at the index of
% that obstacle is set equal to the distance to that obstacle. All
% other elements of dist_array are set to a dummy large distance
% (1e10).

tol = 1e-12;

Pos1 = Pos(1);    Pos2 = Pos(2);
dist1 = sqrt((Obstacles(:,1) - Pos1).^2 + (Obstacles(:,2) - Pos2).^2);
dist2 = sqrt((Obstacles(:,3) - Pos1).^2 + (Obstacles(:,4) - Pos2).^2);
xml = (Obstacles(:,1) + Obstacles(:,3))/2;
yn = (Obstacles(:,2) + Obstacles(:,4))/2;
dist3 = sqrt((xml - Pos1).^2 + (yn - Pos2).^2);
xl = (Obstacles(:,1) + xml)/2;
yl = (Obstacles(:,2) + yn)/2;
xr = (xml + Obstacles(:,3))/2;
yr = (yn + Obstacles(:,4))/2;
dist4 = sqrt((xl - Pos1).^2 + (yl - Pos2).^2);
dist5 = sqrt((xr - Pos1).^2 + (yr - Pos2).^2);
dist = min([dist1 dist2 dist3 dist4 dist5]);
dist_array = 1e10*ones(size(dist));
mins = find(dist<Thresh);

for i = 1:size(mins)
    obs=Obstacles(mins(i),:);
    [xint, yint] = get_intersect(obs, Pos, traj);
    ang = atan2((yint - Pos2), (xint - Pos1));
    rel_ang = abs(traj - ang);
if (rel_ang > pi)  rel_ang = 2*pi - rel_ang;  end;

if (rel_ang < pi/2) & ... 
  (((((obs(3) - xint) > tol) & ((xint - obs(1)) > tol)) | ... 
  (((obs(1) - xint) > tol) & ((xint - obs(3)) > tol))) | ... 
  (((obs(4) - yint) > tol) & ((yint - obs(2)) > tol)) | ... 
  (((obs(2) - yint) > tol) & ((yint - obs(4)) > tol)))
  
  dist_array(mins(i)) = dist(mins(i));
end;

return;
function [dist] = dist_to(obs,Pos1,Pos2)

% This function computes the approximate distance between the MAV and the current obstacle by taking the minimum of the point-to-point distances between the MAV and 5 points along the obstacle. These 5 points are the 2 endpoints of the obstacle, and points at 1/4, 1/2, and 3/4 of its length.

Pos = [Pos1 Pos2];
dist1 = sqrt((obs(1) - Pos(1))^2 + (obs(2) - Pos(2))^2);
dist2 = sqrt((obs(3) - Pos(1))^2 + (obs(4) - Pos(2))^2);
xmid = (obs(1) + obs(3))/2;
ymid = (obs(2) + obs(4))/2;
dist3 = sqrt((xmid - Pos(1))^2 + (ymid - Pos(2))^2);
xleft = (obs(1) + xmid)/2;
yleft = (obs(2) + ymid)/2;
xright = (xmid + obs(3))/2;
yright = (ymid + obs(4))/2;
dist4 = sqrt((xleft - Pos(1))^2 + (yleft - Pos(2))^2);
dist5 = sqrt((xright - Pos(1))^2 + (yright - Pos(2))^2);
dist = min([dist1 dist2 dist3 dist4 dist5]);

return;
function [xint, yint] = get_intersect(obs, Pos, psi)

% function [xint, yint] = get_intersect(obs, Pos, psi)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
% INPUTS:
% obs.................. array of current obstacle information
% (endpoint locations and Voronoi vector directions)
% Pos.................. vector of current coordinates (North, East)
% of MAV (m)
% psi.................. current heading of the MAV (rad)
% OUTPUTS:
% xint.................. North coordinate of intersection point (m)
% yint.................. East coordinate of intersection point (m)
% NOTES:
% This function computes the intersection point of two lines, one passing through both endpoints of obs, and one passing through Pos with a slope of psi. Cramer's Rule is used to calculate the intersection point. If the lines are parallel, a check is performed to see if the lines are coincident, or distinct. If coincident, the intersection point will be the closest endpoint of the obstacle. If distinct, a dummy intersection point is returned (along the line defined by the obstacle, but outside the obstacle endpoints).

if (obs(3) - obs(1)) == 0
    a1 = 1e50;
else
    a1 = -((obs(4) - obs(2)) / (obs(3) - obs(1)));
end;
b1 = 1;
d1 = a1*obs(1) + obs(2);
a2 = -tan(psi);
b2 = 1;
d2 = a2*Pos(1) + Pos(2);

D = det([a1 b1; a2 b2]);
if abs(D) > 1e-12
    xint = det([d1 b1; d2 b2]) / D;
yint = det([a1 d1; a2 d2]) / D;
else % trajectory is parallel to obstacle
    if (abs(a2*obs(1) + obs(2) - d2) < 1e-12) | ...
        (abs(a2*obs(3) + obs(4) - d2) < 1e-12)
        dist1 = sqrt((obs(1) - Pos(1))^2 + (obs(2) - Pos(2))^2);
        dist2 = sqrt((obs(3) - Pos(1))^2 + (obs(4) - Pos(2))^2);
        if dist1 < dist2
            xint = obs(1);  % trajectory is along obstacle, will hit 1st endpoint
            yint = obs(2);
        else
            yint = obs(2);
    end
if (abs(a2*obs(1) + obs(2) - d2) < 1e-12) | ...
        (abs(a2*obs(3) + obs(4) - d2) < 1e-12)
        dist1 = sqrt((obs(1) - Pos(1))^2 + (obs(2) - Pos(2))^2);
        dist2 = sqrt((obs(3) - Pos(1))^2 + (obs(4) - Pos(2))^2);
        if dist1 < dist2
            xint = obs(1);  % trajectory is along obstacle, will hit 1st endpoint
            yint = obs(2);
        else
            yint = obs(2);
    end

-62-
xint = obs(3);  \% trajectory is along obstacle, will hit
\% 2nd endpoint
yint = obs(4);
end;
else  \% trajectory will never hit obstacle, create dummy
\% intersection point
theta = atan2(obs(4) - obs(2), obs(3) - obs(1));
xint = obs(1) - 100*cos(theta);
yint = obs(2) - 100*sin(theta);
end;
end;

return;
function [good_aim] = check_ang(desired_dir,psi,ang_tol)

% function [good_aim] = check_ang(desired_dir,psi,ang_tol)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% desired_dir...........desired heading for the MAV (rad)
% psi.................current heading of the MAV (rad)
% ang_tol...............angular tolerance (rad)
%
% OUTPUTS:
% good_aim............true/false flag that denotes if the MAV is
%.................headed in the desired direction within the
%....................given angular tolerance
%
% NOTES:
% This function is called from the HEADHOLD, HEADHOLD_OBS, and
% COORTURN states.

temp = desired_dir - psi;
if (desired_dir - psi) > pi
    temp = (desired_dir - psi - 2*pi);
end;
if (desired_dir - psi) < -pi
    temp = (desired_dir - psi + 2*pi);
end;

if abs(temp) < ang_tol
    good_aim = 1;
else
    good_aim = 0;
end;

return;
function [Rc] = get_hhRc (Command, psi)

% function [Rc] = get_hhRc (Command, psi)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
% % INPUTS:
% % Command............commanded heading (rad)
% % psi.................current heading of MAV (rad)
% % % OUTPUTS:
% % Rc..............yaw rate command (rad/s)
% % % NOTES:
% % The yaw rate command is proportional to the error between the
% % commanded heading and the current heading, with a saturation
% % limit. This function is called from the HEADHOLD and HEADHOLD_OBS
% % states, where gentle turns are required.

HEADHOLDGAIN = -1.2;

Rc = (Command - psi)*HEADHOLDGAIN;

if (Command - psi) > pi
    Rc = (Command - psi - 2*pi)*HEADHOLDGAIN;
end;
if (Command - psi) < -pi
    Rc = (Command - psi + 2*pi)*HEADHOLDGAIN;
end;

if abs(Rc) > 0.6
    Rc = 0.6*Rc/abs(Rc);
end;

return;
File 10: get_turn_dir.m

function [turn_dir] = get_turn_dir(obs, psi)

% function [turn_dir] = get_turn_dir(obs, psi)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% obs...................array of current obstacle information
% (endpoint locations and Voronoi vector
directions)
% psi...................current heading of the MAV (rad)
%
% OUTPUTS:
% turn_dir..............equal to "1", "2", "-1", or "-2". See NOTES
% for details.
%
% NOTES:
% This function first determines whether the MAV should turn toward
% the first or second endpoint of the current obstacle, depending on
% which is closest to its current heading. If the current obstacle
% is in the MAV's direct path, it is instructed to avoid turning
% towards inside corners, if it can safely do so without executing a
% turn of greater than 135 degrees. turn_dir is set to "1" or "2",
% denoting a turn towards the first or second endpoint, respectively.
% Next, if the current obstacle is in the MAV's line of sight to the
% target, a check is performed to see if the endpoint of interest is
% an outside corner. If so, then turn_dir is set to "-1" or "-2",
% respectively, which will trigger a transition to the HEADHOLD_CORNER
% state, which drives the MAV around the specified outside corner.
% The global variables LOS and HIT are used here. See get_obs.m for
% descriptions.

global LOS HIT;

theta2 = atan2(obs(4) - obs(2), obs(3) - obs(1));
theta1 = atan2(obs(2) - obs(4), obs(1) - obs(3));
tol = 1e-12;

ang1 = abs(theta1 - psi);
if (ang1 > pi) ang1 = 2*pi - ang1; end;
ang2 = abs(theta2 - psi);
if (ang2 > pi) ang2 = 2*pi - ang2; end;

if HIT == 1 & cos(theta2 - obs(5)) > 0.5 & ang2 < 3*pi/4
    turn_dir = 2; % can't turn towards 1st endpoint
    % (<120 deg. inside corner)
elseif HIT == 1 & cos(theta1 - obs(6)) > 0.5 & ang1 < 3*pi/4
    turn_dir = 1; % can't turn towards 2nd endpoint
else
    if ang1 < ang2
        turn_dir = 1;
    else
        turn_dir = 2;
    end;
if abs(psi) == pi
    if abs(theta1) > abs(theta2)
        turn_dir = 1;
    else
        turn_dir = 2;
    end;
end;

if (LOS == 1) & (turn_dir == 1) & (cos(theta2 - obs(5)) < -0.5)
    turn_dir = -1;  % 1st endpoint is outside corner (> 240 deg.)
elseif (LOS == 1) & (turn_dir == 2) & (cos(theta1 - obs(6)) < -0.5)
    turn_dir = -2;  % 2nd endpoint is outside corner
end;

return;
function [Command] = getctCommand(obs, turn_dir)

% function [Command] = getctCommand(obs, turn_dir)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% obs..................array of current obstacle information
% (endpoint locations and Voronoi vector directions)
% turn_dir...............equal to "1" or "2", specifying whether to
% turn in the direction of the first endpoint or
% the second endpoint of the current obstacle,
% respectively
%
% OUTPUTS:
% Command...............heading command for MAV (rad)
%
% NOTES:
% This function is called from the COORTURN state. The Command will
% be parallel to the current obstacle, in the appropriate direction
% (specified by turn_dir).

theta2 = atan2(obs(4) - obs(2), obs(3) - obs(1));
theta1 = atan2(obs(2) - obs(4), obs(1) - obs(3));

if turn_dir == 1
    Command = theta1;
elseif turn_dir == 2
    Command = theta2;
else
    disp ('Error in getctCommand!')
end;

return;
function [Rc] = get_ctRc(Command, psi)

% function [Rc] = get_ctRc(Command, psi)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Command........commanded heading (rad)
% psi...........current heading of the MAV (rad)
%
% OUTPUTS:
% Rc...............yaw rate command (rad/s)
%
% NOTES:
% The yaw rate command is the maximum command that the MAV can
% handle, with the appropriate sign (depending on if a left or right
% turn is necessary to reach the commanded heading). This function
% is called from COORTURN, where sharp turns are required.

global maxRc;

if Command > psi
    if abs(Command - psi) < pi
        Rc = -maxRc; %right
    else
        Rc = maxRc; %left
    end;
else
    if abs(Command - psi) < pi
        Rc = maxRc; %left
    else
        Rc = -maxRc; %right
    end;
end;
return;
function [Ocurr] = get_obs2(Pos1,Pos2,Obstacles,Thresh,psi,...
    targ_dir,Obs)

% function [Ocurr] = get_obs2(Pos1,Pos2,Obstacles,Thresh,psi,...
%    targ_dir,Obs)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Pos1...............current North coordinate of MAV (m)
% Pos2...............current East coordinate of MAV (m)
% Obstacles...........matrix of information for all obstacles in the
%                     environment (endpoint locations and Voronoi
%                     vector directions)
% Thresh.............threshold distance (m). Only obstacles within
%                     the threshold distance of the MAV will be
%                     considered.
% psi................current heading of the MAV (rad)
% targ_dir..........heading from current MAV position to current
% target position (rad)
% Obs................index of current obstacle being avoided
%
% OUTPUTS:
% Ocurr..............index of new obstacle to be avoided. See NOTES
%                    for details.
%
% NOTES:
% This function is called only from HEADHOLD_OBS, where the MAV is
% being commanded to turn parallel to an obstacle (Obs). The MAV's
% direct path must be checked for other obstacles in the way. If one
% is found, the global variable HIT is set to 1, and Ocurr is set to
% the index of this obstacle. LOS is also set appropriately. If the
% new obstacle has the same heading of Obs, then it is ignored.

global LOS HIT;

Ocurr = 0;
Pos = [Pos1 Pos2];
tol = 1e-12;

dist_array = get_dist_array(Obstacles, Pos, psi, Thresh);
Ocurr = find(dist_array == min(dist_array));
Ocurr = Ocurr(1);
if min(dist_array) == 1e10 | Ocurr == Obs % no new obstacles in
    % direct path
    HIT = 0;
    Ocurr = 0;
else % new obstacle in direct path
    theta_Obs = atan2(Obstacles(Obs,4) - Obstacles(Obs,2), ...
    Obstacles(Obs,3) - Obstacles(Obs,1));
    theta_Ocurr = atan2(Obstacles(Ocurr,4) - Obstacles(Ocurr,2), ...
    Obstacles(Ocurr,3) - Obstacles(Ocurr,1));
    Obs_Ocurr = abs(theta_Obs - theta_Ocurr);
    if (Obs_Ocurr > pi) Obs_Ocurr = 2*pi - Obs_Ocurr; end;
end;
if 1 - cos(Obs_Ocurr) < 0.0002  % new obstacle has same heading
    % as Obs, so ignore it
    HIT = 0;
    Ocurr = 0;
else  % new obstacle must be avoided, and LOS must be set
    % appropriately
    HIT = 1;
    obs = Obstacles(Ocurr,:);
    [xint, yint] = get_intersect (obs, Pos, targ_dir);
    ang = atan2((yint - Pos(2)), (xint - Pos(1)));
    rel_ang = abs(targ_dir - ang);
    if (rel_ang > pi) rel_ang = 2*pi - rel_ang; end;
    if (rel_ang < pi/2) ...
        (((((obs(3) - xint) > tol) & ((xint - obs(1)) > tol)) | ...
        (((obs(1) - xint) > tol) & ((xint - obs(3)) > tol))) | ...
        (((obs(4) - yint) > tol) & ((yint - obs(2)) > tol)) | ...
        (((obs(2) - yint) > tol) & ((yint - obs(4)) > tol)))
        LOS = 1;  % same obstacle also in LOS to target
    else
        LOS = 0;
    end;
end;

return;
function [Command] = get hhCommand(Pos1, Pos2, obs, turn_dir)

% function [Command] = get hhCommand(Pos1, Pos2, obs, turn_dir)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
% INPUTS:
% Pos1..................current North coordinate of MAV (m)
% Pos2..................current East coordinate of MAV (m)
% obs..................array of current obstacle information
% (endpoint locations and Voronoi vector directions)
% turn_dir.............equal to "-1" or "-2", specifying whether the first endpoint or the second endpoint of the current obstacle is the outside corner of interest, respectively
% OUTPUTS:
% Command..............heading command for MAV (rad)
% NOTES:
% This function is called from HEADHOLDCORNER, which drives the MAV around an outside corner of the given obstacle. This is accomplished by setting the heading command to be in the direction of the "aimpoint", a point L meters out from the corner of interest, along the Voronoi vector.

L = 5;
if turn_dir == -1
    aimpoint = [obs(1)+L*cos(obs(5)), obs(2)+L*sin(obs(5))];
    Command = atan2 ((aimpoint(2) - Pos2), (aimpoint(1) - Pos1));
elseif turn_dir == -2
    aimpoint = [obs(3)+L*cos(obs(6)), obs(4)+L*sin(obs(6))];
    Command = atan2 ((aimpoint(2) - Pos2), (aimpoint(1) - Pos1));
else
    disp('Error in get hhCommand!')
end;
return;
function [Rc] = get_hhRc_corner (Command, psi)

% function [Rc] = get_hhRc_corner (Command, psi)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
% INPUTS:
% Command .......... commanded heading (rad)
% psi ................ current heading of MAV (rad)
% OUTPUTS:
% Rc ................ yaw rate command (rad/s)
% NOTES:
% The yaw rate command is proportional to the error between the
% commanded heading and the current heading, with a saturation
% limit. This function is called from the HEADHOLD_CORNER state,
% where gentle turns are required.
% This file is actually identical to get_hhRc, but is kept separate
% for purposes of experimenting with a different gain or limit.

HEADHOLDGAIN = -1.2;

Rc = (Command - psi)*HEADHOLDGAIN;

if (Command - psi) > pi
    Rc = (Command - psi - 2*pi)*HEADHOLDGAIN;
end;
if (Command - psi) < -pi
    Rc = (Command - psi + 2*pi)*HEADHOLDGAIN;
end;

if abs(Rc) > 0.6
    Rc = 0.6*Rc/abs(Rc);
end;

return;
function [out_of_sight] = get_oos(Pos1,Pos2,obs,targ_dir)

% function [out_of_sight] = get_oos(Pos1,Pos2,obs,targ_dir)
% AUTHOR: Rebecca J. Dailey
% DATE: 10/01/01
%
% INPUTS:
% Pos1........................current North coordinate of MAV (m)
% Pos2........................current East coordinate of MAV (m)
% obs..........................array of current obstacle information
% (endpoint locations and Voronoi vector directions)
% targ_dir....................heading from current MAV position to
% current target position (rad)
%
% OUTPUTS:
% out_of_sight...............true/false flag that indicates if the
% current obstacle is out of the line of
% sight from the MAV to the target
%
% NOTES:
% This function is called from HEADHOLD_CORNER, which uses
% out_of_sight as part of its exit criteria.

tol = 1e-12;
Pos = [Pos1 Pos2];
[xint, yint] = get_intersect(obs, Pos, targ_dir);
ang = atan2((yint - Pos(2)), (xint - Pos(1)));
rel_ang = abs(targ_dir - ang);
if (rel_ang > pi) rel_ang = 2*pi - rel_ang; end;

if (rel_ang < pi/2) & ...
    (((((obs(3) - xint) > tol) & ((xint - obs(1)) > tol)) | ...)
    (((obs(1) - xint) > tol) & ((xint - obs(3)) > tol))) | ...
    (((obs(4) - yint) > tol) & ((yint - obs(2)) > tol)) | ...
    (((obs(2) - yint) > tol) & ((yint - obs(4)) > tol)))
    out_of_sight = 0;
else
    out_of_sight = 1;
end;
return;