WestVirginiaUniversity
THE RESEARCH REPOSITORY @ WVU

Graduate Theses, Dissertations, and Problem Reports

2004

Decentralized control for UAV path planning and task allocation

Matthew C. Lechliter
West Virginia University

Follow this and additional works at: https://researchrepository.wvu.edu/etd

Recommended Citation

Lechliter, Matthew C., "Decentralized control for UAV path planning and task allocation" (2004). *Graduate Theses, Dissertations, and Problem Reports*. 1443.
https://researchrepository.wvu.edu/etd/1443

This Thesis is protected by copyright and/or related rights. It has been brought to you by the The Research Repository @ WVU with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you must obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/ or on the work itself. This Thesis has been accepted for inclusion in WVU Graduate Theses, Dissertations, and Problem Reports collection by an authorized administrator of The Research Repository @ WVU. For more information, please contact researchrepository@mail.wvu.edu.

Decentralized Control for UAV Path Planning and Task Allocation

Matthew C. Lechliter

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Aerospace Engineering

Marcello R. Napolitano, Ph.D., Chair
Gary Morris, Ph.D.
Jacky Prucz, Ph.D.

Department of Mechanical and Aerospace Engineering

West Virginia University
Morgantown, WV
2004

Keywords: unmanned air vehicle, cooperative control, decentralized control

ABSTRACT

Decentralized Control for UAV Path Planning and Task Allocation

Matthew C. Lechliter

The effort of this research is to move toward enabling Unmanned Air Vehicles to fly in autonomous formations with intelligent mission planning capabilities. In particular, UAVs will be able to autonomously perform path planning and task allocation. During missions, the UAVs must be able to avoid threats and no-fly zones while still reaching their target optimally in time.

A path planning and task allocation approach was first developed that treats the problem as a *Multi-dimensional, Multiple-Choice Knapsack Problem*. Paths are selected and task assigned while minimizing the UAV team's overall mission cost. Next, a SIMULINK-based centralized simulation environment was created. This simulation uses the path planning and task allocation scheme previously developed, and adds time-varying, dynamic environment aspects. The latter part of the research effort was focused on development of a decentralized simulation environment. This decentralized version includes a vehicle's own decision making capabilities and communication amongst a team of vehicles.

The decentralized simulation was compared with the centralized version in terms of simulation efficiency and was found to be faster for individual UAVs. Finally, real communications issues were addressed to show that while communication problems lead to a lack of cooperation, tasks can still be performed and missions completed within the decentralized simulation environment.

Acknowledgments

I would first like to thank my wife Leah for all her love and support throughout these last two years. Your dedication to me while obtaining this degree and choosing my career path means more to me than I can express. I love you.

I would like to thank my committee chairman and research advisor Dr. Marcello Napolitano. Your help and guidance throughout the last two years have been integral to not only this research, but my career as well.

I would like to acknowledge and thank my committee members Dr. Garry Morris and Dr. Jacky Prucz for taking time from their busy schedules to review and contribute their thoughts to this research effort.

I would like to thank everyone who contributed to this research effort: Zachary Spritzer, Jennifer Hazelton, Dr. Giampiero Campa, Dr. Brad Seanor, Elena Lucci, and Dr. Mario George Perhinschi.

Finally, I would like to thank God, for through Him all things are possible.

Table of Contents

Title Page i
Abstract ii
Acknowledgments iii
Table of Contents iv
List of Tables vi
List of Figures vii
Nomenclature x
Chapter 1: Introduction 1
1.1 UAV History 1
1.2 Envisioned Future 4
1.3 Research Objectives 8
Chapter 2: Literary Review 11
2.1 Path Planning Methods 11
2.2 Path Planning/Task Allocation Approaches 14
2.3 Decentralized Control and Communications 17
Chapter 3: Development of the Path Planning/Task Allocation Scheme 18
3.1 Discussion of Setup 18
3.2 Voronoi Diagram Generation 20
3.3 Dijkstra's Algorithm and Cost Assignment 23
3.4 Path Shortening and Flyability 28
3.5 Multi-dimensional, Multiple-Choice Knapsack Problem 38
Chapter 4: Aircraft Dynamics 41
4.1 Introduction 41
4.2 Body Axes Modeling 42
4.3 Flight Path Equations 47
4.4 Earth-fixed Axes and Kinematic Relationships 51
Chapter 5: Development of Centralized UAV Simulation 55
5.1 Main Simulation System 55
5.2 Simulation Inputs 56
5.3 Path Planning and Task Allocation Execution 64
5.4 Aircraft Dynamics Subsystem 67
5.5 UAVs Manager 76
5.6 Targets Manager 81
5.7 Threats Manager 86
5.8 Simulation Outputs 88
Chapter 6: Decentralized Path Planning and Task Allocation 101
6.1 Main Simulation System 101
6.2 Individual UAV System 102
6.3 UAV Communications 103
6.4 Individual UAV Calculations 104
6.5 Simulation Outputs 109
Chapter 7: Comparison of Decentralized and Centralized Simulations 112
7.1 Simulation Efficiency 112
7.2 Miscommunication 122

7.3 Delay of Communication 126
7.4 Loss of Communication 127
Chapter 8: Conclusions and Recommendations 132
8.1 Conclusions 132
8.2 Recommendations 133
References 134
Appendix A: MATLAB codes for Path Planning and Task Allocation 139
path_planning.m 140
filter_zeros.m 142
vrn_diag_gen.m 143
voronoi.m 145
connect_vrn.m 148
cheapest_paths.m 149
set_THC.m 151
c_assign.m 152
list2adj.m 154
adj2list.m 156
pred2path.m 157
mat2vec.m 159
isint.m 160
dijk.m 161
path_shrtng.m 164
shorten_paths.m 167
fillet_path.m 169
heading_angle_paths.m 170
update_cost.m 174
mmkp_task_allocation.m 175
mmkp_new.m 176
vrt_sim_convert.m 177
plot_uav.m 179
Appendix B: MATLAB codes for Simulation 181
place_waypoints_s.m 182
place_waypoints.m 183
path_planning_s.m 185
uav_crash_s.m 186
uav_crash.m 187
uav_intercepted_s.m 188
uav_intercepted.m 189
target_classifier_s.m 190
target_classifier.m 191
compare_targets_s.m 193
compare_targets.m 194
compare_threats_s.m 195
compare_threats.m 196
display_initial_s.m 197
display_initial.m 198

List of Tables

Table 3.1: Typical threats 24
Table 3.2: List of example path permutations and mission costs 39
Table 7.1: Summary of MATLAB Profile Reports 112
Table 7.2: Profile Report based on 4 UAVs, 4 Targets, 4 Threats, and 4 No-fly Zones 113
Table 7.3: Profile Report based on 5 UAVs, 5 Targets, 5 Threats, and 5 No-fly Zones 114
Table 7.4: Profile Report based on 9 UAVs, 9 Targets, 15 Threats, and 15 No-fly Zones ...115
Table 7.5: SIMULINK Profile Summary for centralized simulation 118
Table 7.6: SIMULINK Profile Report for centralized version 119
Table 7.7: SIMULINK Profile Report for centralized version, with Accelerator 119
Table 7.8: SIMULINK Profile Summary for decentralized simulation 120
Table 7.9: SIMULINK Profile Report for decentralized version 120
Table 7.10: SIMULINK Profile Report for decentralized version, with Accelerator 121

List of Figures

Figure 1.1: USAF *Firebee* drone 2
Figure 1.2: U.S. MQ-1 *Predator* UAV, equipped with Hellfire missiles 3
Figure 1.3: U.S. Air Force RQ-4 *Global Hawk* 4
Figure 1.4: Department of Defense Annual Funding Profile for UAVs 5
Figure 1.5: Autonomous Control Level Trend 6
Figure 1.6: LOCAAS mini-UAV munition 7
Figure 1.7: U.S. Air Force X-45A UAV 8
Figure 1.8: Cooperative Operation of UAVs for SEAD 9
Figure 2.1: Vertices of a simple graph 11
Figure 2.2: Voronoi diagram for threat locations 12
Figure 3.1: Voronoi diagram with 25 sites 21
Figure 3.2: Crotale "Rattlesnake" surface-to-air missile 25
Figure 3.3: Example directed graph with costs 27
Figure 3.4: Picture illustrating fillet principle 32
Figure 3.5: Example of heading angle solution 34
Figure 3.6: Second example of heading angle solution 35
Figure 3.7: Final example of heading angle solution 35
Figure 3.8: Example UAV to target MMKP setup 39
Figure 4.1: Body axis system with forces and moments 42
Figure 4.2: Stability axis system and angles with body axis system 47
Figure 4.3: Aircraft orientation with Euler angles 52
Figure 5.1: Main simulation system 55
Figure 5.2: Cooperating UAVs Simulation Main Menu 57
Figure 5.3: Error message 58
Figure 5.4: Aircraft Menu GUI 59
Figure 5.5: Point-and-click method of placing UAV positions 60
Figure 5.6: Pop-up Target Menu 61
Figure 5.7: No-Fly Zones Menu 62
Figure 5.8: Pop-up Threats Menu 63
Figure 5.9: Example battlefield setup 64
Figure 5.10: Place Waypoints block 65
Figure 5.11: Path Planning and Task Allocation block 66
Figure 5.12: 'UAV Dynamics' blocks for all UAVs 68
Figure 5.13: Blocks to output UAV positions, heading angle, and signal end of path 69
Figure 5.14: Determining next position in path, runs aircraft model, and signal end of path 69
Figure 5.15: Blocks that 'look ahead' and output next position in path 70
Figure 5.16: Determination of end of assigned path 70
Figure 5.17: Actual UAV dynamics block, with aircraft model and heading-angle autopilot 71
Figure 5.18: Flight simulation environment for aircraft model 72
Figure 5.19: Parameters and inputs for aircraft model 73
Figure 5.20: Actuator and cable dynamics subsystem 74
Figure 5.21: Heading angle autopilot, showing turn generator 75
Figure 5.22: Turn generator subsystem 75
Figure 5.23: UAV Positions block 76

Figure 5.24: UAV CRASH block 77
Figure 5.25: UAV INTERCEPTED block.....77
Figure 5.26: UAV DOWN block 78
Figure 5.27: UAV MANAGER subsystem 79
Figure 5.28: Individual UAV manager for tracking positions, velocity, and destruction 80
Figure 5.29: Printing blocks for UAV destruction.....81
Figure 5.30: Target State Manager 82
Figure 5.31: Target classifier function.....83
Figure 5.32: Part of target classification used for signaling replan 83
Figure 5.33: TARGETS MANAGER.....84
Figure 5.34: Part of target management used for signaling replan.....84
Figure 5.35: Pop-up target manager.....85
Figure 5.36: Pop-up target manager for an individual target.....85
Figure 5.37: THREATS MANAGER.....86
Figure 5.38: Part of threat management used for signaling replan.....86
Figure 5.39: THREAT CHANGE blocks 87
Figure 5.40: Pop-up and firing threat manager for an individual threat 88
Figure 5.41: Initial battlefield setup.....89
Figure 5.42: Path Planning and Task Allocation occurring at time 0 90
Figure 5.43: Path Planning and Task Allocation occurring at time 100 91
Figure 5.44: Path Planning and Task Allocation occurring at time 150 91
Figure 5.45: Path Planning and Task Allocation occurring at time 325 92
Figure 5.46: Detail of UAV 3 turning to now attack target 1 at time 325 92
Figure 5.47: Path Planning and Task Allocation occurring at time 462 93
Figure 5.48: Path Planning and Task Allocation occurring at time 538 93
Figure 5.49: Path Planning and Task Allocation occurring at time 688 94
Figure 5.50: Path Planning and Task Allocation occurring at time 704 94
Figure 5.51: Path Planning and Task Allocation occurring at time 749 95
Figure 5.52: Path Planning and Task Allocation occurring at time 764 95
Figure 5.53: Path Planning and Task Allocation occurring at time 838 96
Figure 5.54: Path Planning and Task Allocation occurring at time 878 96
Figure 5.55: Path Planning and Task Allocation occurring at time 921 97
Figure 5.56: Path Planning and Task Allocation occurring at time 938 97
Figure 5.57: Path Planning and Task Allocation occurring at time 978 98
Figure 5.58: Path Planning and Task Allocation occurring at time 1014 98
Figure 5.59: Path Planning and Task Allocation occurring at time 1056 99
Figure 5.60: Path Planning and Task Allocation occurring at time 1098 99
Figure 6.1: Main simulation system for decentralized UAV control.....101
Figure 6.2: Main system for individual UAVs 102
Figure 6.3: ‘UAV Dynamics’ blocks for UAV 1.....105
Figure 6.4: UAV Positions block.....106
Figure 6.5: Individual UAV MANAGER subsystem 106
Figure 6.6: UAV initialization block with UAV REPLAN subsystem 107
Figure 6.7: UAV REPLAN subsystem 107
Figure 6.8: TARGETS MANAGER.....108
Figure 6.9: TARGETS initialization block with UAV REPLAN subsystem.....108

Figure 6.10: TARGET REPLAN subsystem 108
Figure 6.11: Initial battlefield setup for decentralized simulation example 110
Figure 6.12: Decentralized simulation example 111
Figure 7.1: Initial battlefield setup for SIMULINK Profile Reports 118
Figure 7.2: Main system for decentralized UAV control with miscommunication.....123
Figure 7.3: NOISE block used for simulating miscommunication.....124
Figure 7.4: Individual UAV noise 124
Figure 7.5: Initial battlefield setup for miscommunication example.....125
Figure 7.6: Miscommunication, decentralized simulation example 126
Figure 7.7: Main system with individual UAV communication loss 128
Figure 7.8: Main system for individual UAV 2, showing modifications 129
Figure 7.9: Loss of team of UAVs block.....129
Figure 7.10: Initial battlefield setup for individual communication loss example 130
Figure 7.11: Individual communication loss example.....131

Nomenclature

Symbol	Description
English	
C_D	Drag coefficient
C_L	Lift coefficient
C_l	Rolling moment coefficient
C_m	Pitching moment coefficient
C_n	Yawing moment coefficient
C_Y	Side force coefficient
H	Altitude
I_x	Airplane moment of inertia about x
I_{xy}	Airplane product of inertia about x
I_{xz}	Airplane product of inertia about z
I_y	Airplane moment of inertia about y
I_{yz}	Airplane product of inertia about y
I_z	Airplane moment of inertia about z
m	Mass
ntarg	Number of targets
nthreats	Number of threats
nuav	Number of UAVs
nzones	Number of no-fly zones
p	Airplane angular velocity component about x
q	Airplane angular velocity component about y
r	Airplane angular velocity component about z
u	Airplane velocity component about x
V	True aircraft velocity
v	Airplane velocity component about y
w	Airplane velocity component about z
x_e	X-position with respect to Earth-fixed axes
y_e	Y-position with respect to Earth-fixed axes
Greek	
α	Angle of attack
α	Angle formed by two intersecting edges
β	Sideslip angle
ψ	Airplane heading angle
θ	Airplane pitch attitude angle
ϕ	Airplane bank angle
Acronym	
GUI	Graphical User Interface
MILP	Mixed-Integer Linear Program
MMKP	Multi-dimensional, Multiple Choice Knapsack Problem
UAV	Unmanned Air Vehicle

Chapter 1
Introduction

1.1 UAV History

The United States Armed Forces has a long history of involvement with Unmanned Air Vehicles (UAVs), with roots beginning in late World War I. The first person to successfully address the issues of automatic stabilization, control, and navigation in creating a UAV was Elmer Ambrose Sperry. In early World War I, the U.S. Navy had appointed him to chair the development of an 'aerial torpedo.' The first successful flight of a UAV occurred on 6 March 1918, when the *Curtis Sperry Aerial Torpedo* was catapulted into the air, flew a preplanned 1000-yard flight, and successfully landed in the waters off Long Island to be later reflown[1]. Other aerial torpedoes soon appeared, including the *Liberty Eagle 'Kettering Bug'*, which attempted to navigate to a target some 50 miles away, turn its engine off, and hit the target with a 200-pound bomb.

The first robotic aircraft to successfully take off, fly radio-controlled maneuvers, and land was the British RAE 1921 *TARGET*, followed a year later by the U.S. Navy's Curtiss N-9 Seaplane on 15 September 1924. The N-9 was remotely controlled for 40 minutes and executed 50 commands before landing[1]. As a result of these early aerial torpedoes efforts, target drones came about in the 1930s. These drones were used to train aerial gunners. The first operation cruise missiles (formerly called aerial torpedoes) were the German V-1 'Buzz Bombs,' which sadly introduced the general public to these weapons, as all previous aerial torpedoes/cruise missiles had been classified. During the course of World War II, some 10,500 V-1s were launched, with over 2,400 reaching their targets, most of which resided in England[1].

Reconnaissance drones were first evaluated in the 1950s. In 1955, the U.S. Army's SD-1 *Observer* became the first tactical UAV. Other reconnaissance drones that appeared during that decade include the Army's SD-2 *Overseer*, SD-3 *Sky Spy*, SD-4 *Swallow*, SD-5 *Osprey*, the U.S. Air Force's GAM-67 *Crossbow*, and the USMC's small

Bikini UAV. However, during the Cuban Missile Crisis of the early 1960's, the Air Force successfully modified some of its Ryan *Firebee* drones to carry cameras and return with reconnaissance pictures. These reconnaissance drones were successfully used in 3,500 sorties flown during the Vietnam Conflict[1].

Figure 1.1: USAF *Firebee* drone (U.S. Air Force photo)[2]

The strike role of UAVs was first explored in 1962 with the U.S. Navy's *Gyrodyne* QH-50 drone helicopter. These unmanned helicopters carried anti-submarine torpedoes. In 1972, the Air Force again modified *Firebee* drones to carry Maverick and Stubby Hobo missiles for use in Suppression of Enemy Air Defenses (SEAD) roles. The end of the Vietnam Conflict, however, put an end to this "Have Lemon" program.

UAV development continued in the 1980's, but really expanded in the 1990's. In the U.S. military's arsenal during this time were the *Predator*, *Hunter*, *Pioneer*, and *Shadow* UAVs, which were used for reconnaissance in the conflicts in the Persian Gulf, the Balkans, and more recently in Afghanistan and Iraq[2]. The MQ-1 (formerly RQ-1)

Predator is a 2,250 pound UAV that has been used by the military forces since 1995. The UAV was used for reconnaissance purposes in Bosnia, Kosovo, Afghanistan, and Iraq with its 24-hour endurance flight time while carrying up to a 450-pound payload. In 2001, a *Predator* was equipped with Hellfire missiles and successfully used to engage targets, thus earning it a multi-mission capability status.

Figure 1.2: U.S. MQ-1 *Predator* UAV, equipped with Hellfire missiles[2]

The RQ-2 *Pioneer* was developed in 1986. It is a Navy UAV that was used in 1991 in the Persian Gulf, as well as in Bosnia and Kosovo. The RQ-5 *Hunter* was used in 1999 through 2002 in NATO operations in the Balkans. The RQ-7 *Shadow* is a U.S. Army UAV. It can provide video surveillance for 4 hours and up to a 50-kilometer range, while carrying a 60-pound payload.

The last of the currently employed UAVs is the Air Force's RQ-4 *Global Hawk*. This is a large 26,750 pound UAV capable of 32-hour flight endurance while carrying a payload of 1950-pounds. It is a high altitude, long endurance UAV designed to provide reconnaissance coverage of up to 40,000 nm^2 per day[2].

Figure 1.3: U.S. Air Force RQ-4 *Global Hawk* (U.S. Air Force photo) [2]

1.2 Envisioned Future

During the decade of the 1990s, the Department of Defense spent roughly $3 billion on Unmanned Air Vehicles. For the following decade, the DOD is scheduled to spend over $10 billion on UAVs! As described in the *Unmanned Aerial Vehicles Roadmap 2002 – 2027*[2], the DOD is aggressively pursuing UAV technology and significantly increasing spending on UAVs. Figure 1.4 illustrates this tremendous increasing in the funding.

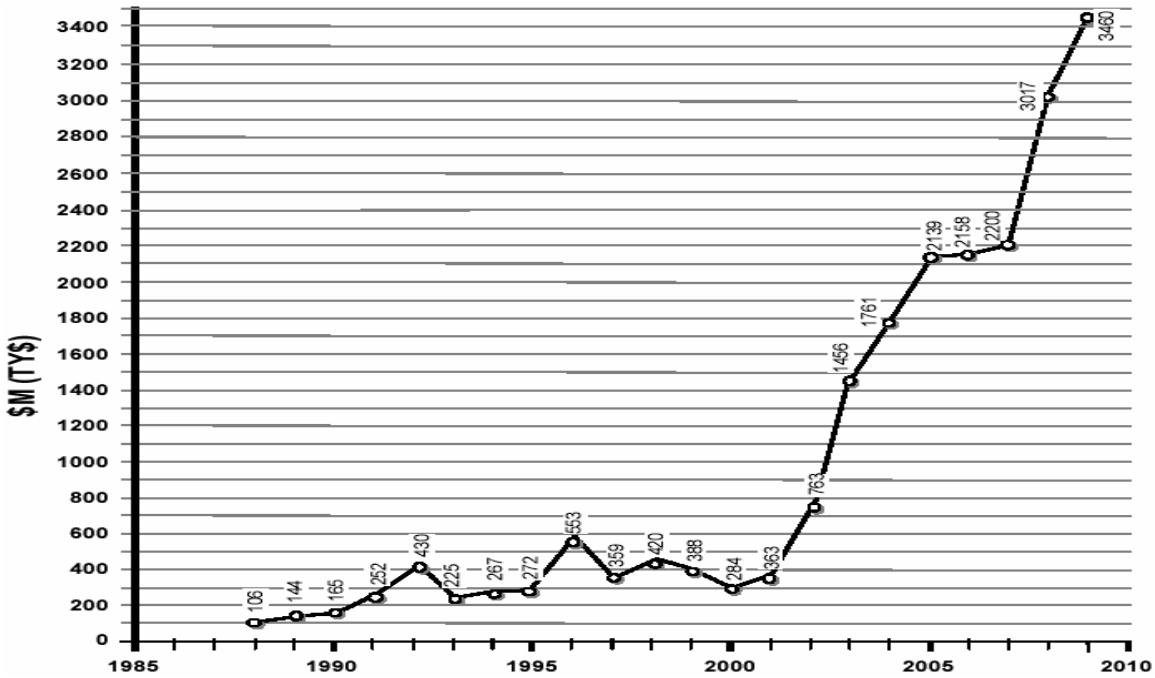

Figure 1.4: Department of Defense Annual Funding Profile for UAVs[2]

UAVs offer numerous advantages to the military. Most notable are the advantages of the ability to perform missions classified as "dull, dirty, or dangerous"[3]. Missions that are classified as dull include examples of an aircraft loitering over airspace for long periods of time while providing surveillance or jamming enemy electronic devices. These types of missions can last for especially long periods of time, such that manned crews would not be optimal to perform, plus UAVs could be outfitted with multiple sensors and/or jamming equipment and provide and even higher efficiency at performing the 'dull' missions. The second type of mission is the dirty type. This type of mission includes reconnaissance in areas that have been contaminated by nuclear, biological, or chemical weapons, where the presence of manned aircraft would put the crew in danger. The last type is the dangerous mission, such as high-risk but high-value targets or Suppression of Enemy Air Defenses (SEAD).

Additional advantages offered by the use of UAVs offer include:

- Maximizing maneuverability, where there are no constraints based on the crew's physical limits;
- Low or no risk to human operators, such as in the dirty or dangerous missions;

- Lower overall weight of the aircraft, resulting from elimination of crew support hardware;
- A lower overall cost, due in part to the lack of crew support hardware and the elimination of expensive pilot training[4].

Currently UAVs require several operators on the ground for control of a single UAV, as all of the current UAVs discussed in Section 1.1 are controlled in this manner. While such elimination of the pilot and crew from the aircraft do result in many benefits such as decreasing cost and eliminating danger to aircrews, the future of UAVs is moving in the direction of autonomy[5]. Autonomous UAVs will require little or no human support to carry out missions, and this addition of autonomy adds another benefit – that is superior coordination among a group of UAVs. Figure 1.5 illustrates the trend in the increase of UAV autonomous control from early in their history until the year 2015.

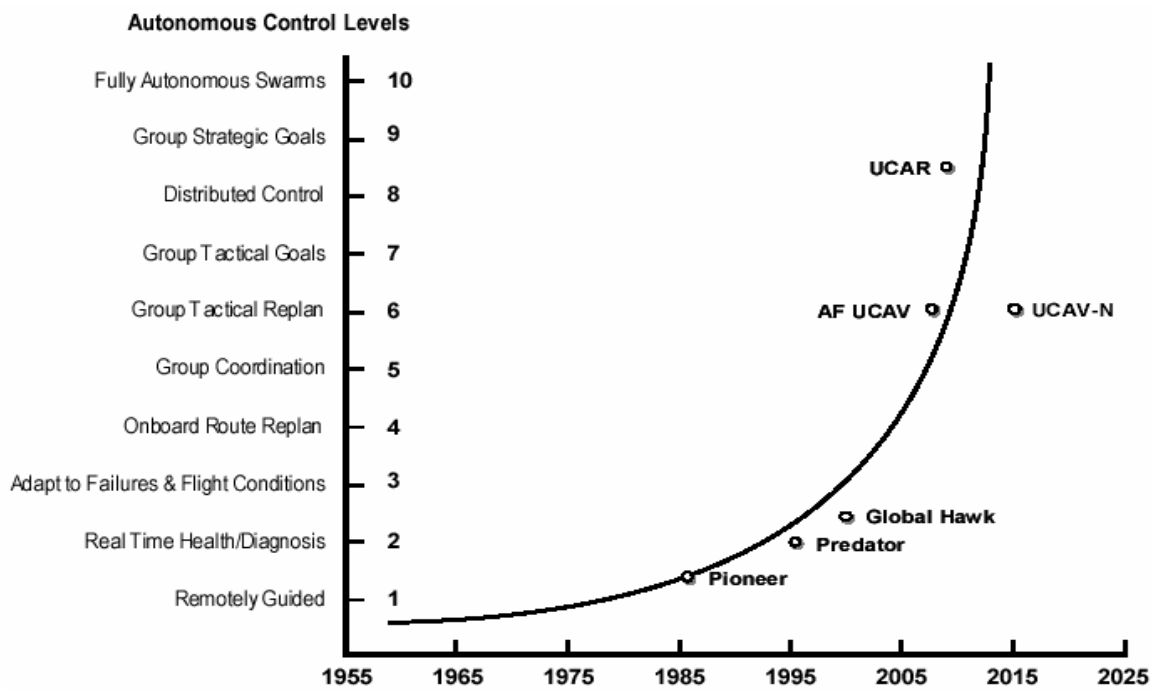

Figure 1.5: Autonomous Control Level Trend (U.S. Air Force)[2]

Cooperative UAV flight based on autonomous aircraft offers capabilities of the use a formation to overwhelm enemy defenses, the ability of adjust timing in a coordinated

attack, and the expansion from the small area a single UAV can see and detect to a much broader situational awareness created by multiple UAVs sharing information[2]. These teams of UAVs lead to superior abilities to perform a large variety of missions, including reconnaissance, jamming, suppression of enemy air defenses, missile defense, fixed and moving high-priority target attacks, and eventually air-to-air combat[4].

Currently there are several DOD projects attempting to address the possibilities of autonomous capabilities for the future for the next quarter-century. These include the *Broad Area Maritime Surveillance*, the RQ-8 *Fire Scout*, the MQ-9 *Predator B*, which is an extension of the current MQ-1 *Predator* to allow hunter-killer groups, the *Dragon Eye* mini-UAV, the *Force Protection Aerial Surveillance System (FPASS)*, *Neptune*, the *Low Cost Autonomous Attack System (LOCAAS)*, and finally the Air Force's *X-45*. The first of significant interest is the LOCAAS. This UAV is a miniature, autonomous munition that is capable of a broad area search, identification, and destruction of ground targets[6]. These UAVs are designed to cooperate upon locating a possible target, and they work together to destroy it, as each is itself also a flying munition. Figure 1.6 illustrates the LOCAAS munition.

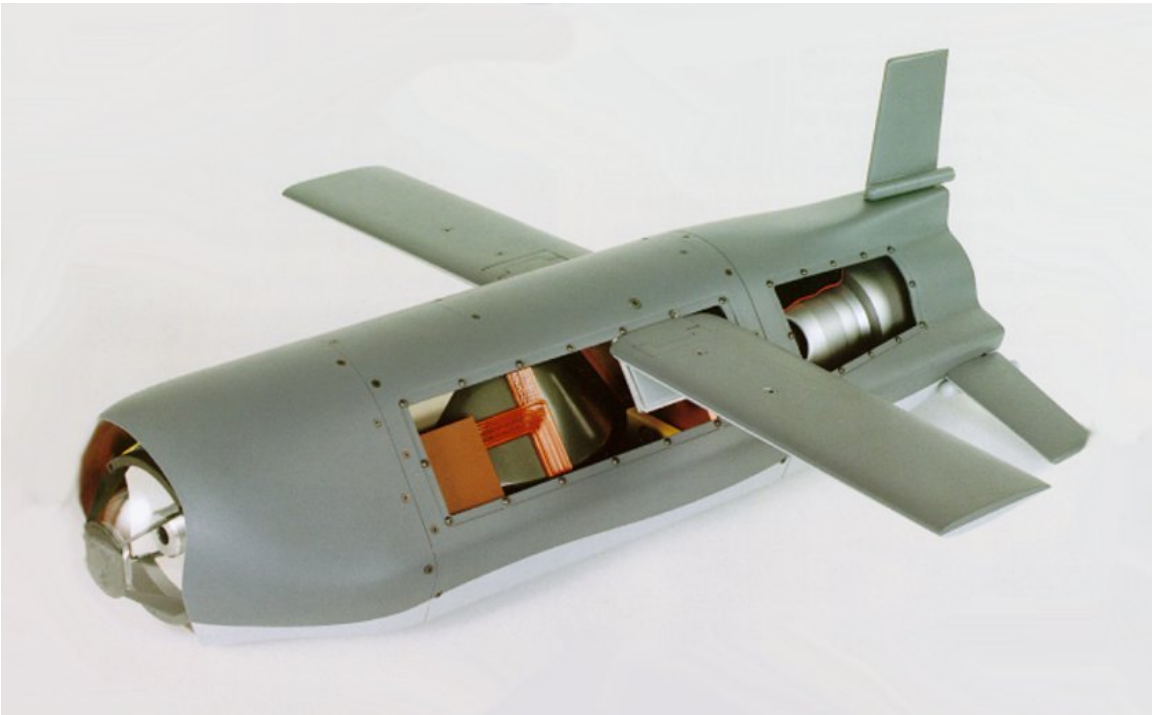

Figure 1.6: LOCAAS mini-UAV munition (U.S. Air Force photo) [2]

Another developmental UAV of interest is the U.S. Air Force's X-45. This Unmanned Combat Air Vehicle (UCAV) is designed to use UAV autonomy and cooperation to perform dangerous but high-priority missions such as high-value targets or SEAD[7]. These UCAVs will be designed to have preprogrammed objectives and target information from ground mission planners. This information is used to carry out missions autonomously and efficiently by taking advantage of cooperation amongst a group.

Figure 1.7: U.S. Air Force X-45A UAV (U.S. Air Force photo)[2]

1.3 Research Objectives

As mentioned in Section 1.2, dangerous missions including Suppression of Enemy Air Defenses (SEAD) and high-risk but high-value target missions are important objectives for future UAV capabilities. These UAVs are very attractive in that they eliminate risk to the human crew while performing these dangerous missions, the aircraft have potential for greater survivability, they have greater endurance to perform a mission as opposed to crew fatigue, the cooperative nature gives a greater probability of successful outcome, and finally cost is reduced[4]. Figure 1.8 illustrates what a typical SEAD or high-risk but high-value might look like, with several cooperating UCAVs attacking targets.

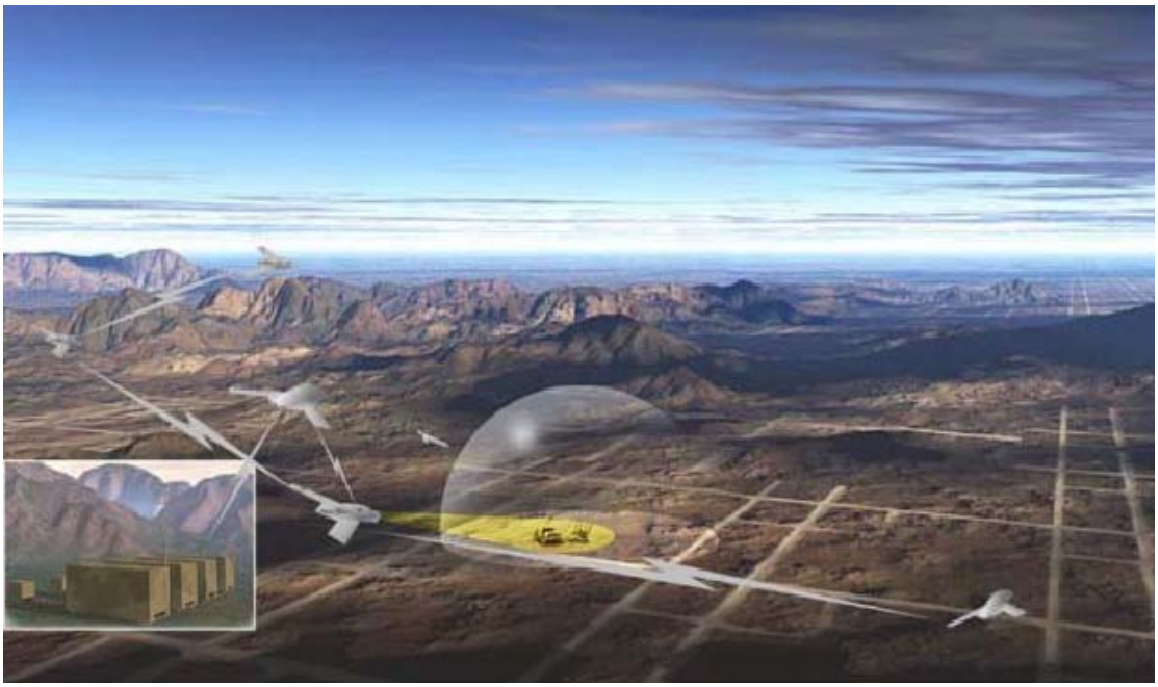

Figure 1.8: Cooperative Operation of UAVs for SEAD (U.S. Air Force picture) [2]

The general basic problem formulation for SEAD or high-risk but high-value missions is as follows: given '*nuav*' UAVs with '*nzones*' no-fly zones such as mountains or political boundaries, and given '*ntarg*' targets or waypoints to visit, the UAVs must accomplish a mission such as visiting each target or waypoint while minimizing an overall cost to the group. Extending this basic formulation to add realistic constrains and boundary conditions include timing constraints, such as a preliminary target needing to be reconnoitered prior to attacking. Also dynamic constraints on planned paths, such as maximum linear velocities for UAVs and maximum angular rates for rolling performance need to be accounted for. Furthermore, the problem may be time varying, where there are addition/removal or targets, loss of UAVs in the team, and loss of communications may occur. Also, in the role of high-risk but high-value missions, there will also be '*nthreats*' threats in the scenario that the UAVs should avoid.

The following research objectives are intended to address the problem of Suppression of Enemy Air Defenses or high-risk but high-value mission planning.

Item #1. A path planning and task allocation scheme must be created for an elementary two-dimensional scenario, with a limited number of UAVs, targets, and no-fly zones. The generated trajectories must be of minimal length, but subject to a cost factor to include flying around the no-fly zones. The trajectories must be dynamically feasible, and additionally, the software must be computationally efficient in order to be run 'real-time'[8].

Item #2. The coding is to be extended to encompass a larger number of UAVs, targets, and no-fly zones, and now has the addition of threats – areas that can be flown into but with an additional cost of the probability of the UAV being destroyed.

Item #3. After the path planning and task allocation scheme is finished, the development of a SIMULINK-based centralized simulation environment is next. This centralized simulation environment is such that a central processor controls all of the decision making abilities for the entire UAV team.

Item #4. After the basic simulation is in place, it now needs to be extended to include the time-varying aspects of the problem. Included in this are 'pop-up' threats, ones that were not previously known to the team of UAVs but appear some time into the mission, varying states of targets, such as 'identified but not reconned,' 'reconned but not attacked,' 'attacked but not confirmed,' and 'confirmed destroyed,' the ability of threats to attempt to destroy UAVs if the UAVs pass within range of the threat, and finally the ability of the group to replan if any of these events occur.

Item #5. Once the time-varying centralized simulation environment is complete, a decentralized simulation environment is to be developed based on the centralized version. This decentralized version now includes a vehicle's own decision making capabilities and communication amongst vehicles.

Item #6. Finally, the decentralized simulation is to be compared to the centralized simulation in terms of 'real-time' efficiency, and the real-life 'what-if' communication problems are to be tested in the decentralized simulation environment.

Chapter 2
Literary Review

2.1 Path Planning Methods

Vehicle path planning is a broad subject with a significant body of research already established, especially in the field of robotics. Applied to UAVs, however, path planning has been the subject of study for only a limited number of years. In general, three different approaches have been studied to generate UAV paths, as discussed by Bortoff[8]. These include graph-based methods, where paths are generated from a sequence of edges connecting vertices of the graph, optimal control, which computes an optimal path based on a cost function, and finally virtual potential fields, where a simpler, related problem is solved to obtain the path[8].

For UAV trajectory planning, graph-based approaches have received the most attention. In a graph approach, vertices are assigned to discrete points in space, edges are used to connect these vertices, costs are assigned to each of the edges, and lastly the graph is searched for an optimal trajectory[8]. For a simple graph, vertices can be assigned rectangular points, as illustrated in Figure 2.1.

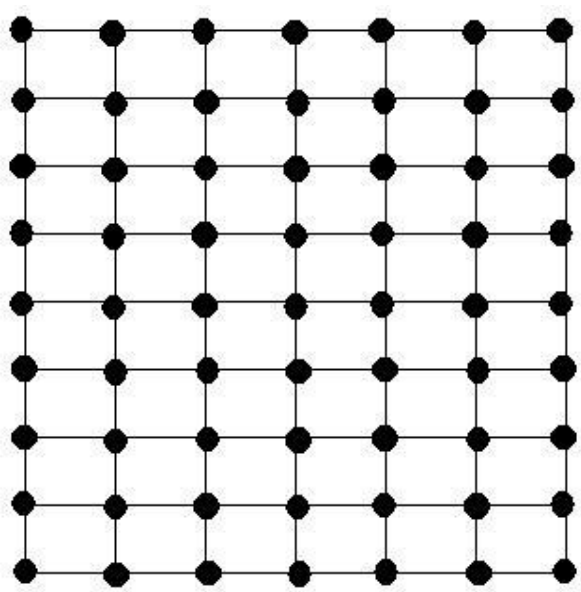

Figure 2.1: Vertices of a simple graph

However, in this simple arrangement, for a well-defined graph, the computational complexity tends to grow at an exponential rate. A graph with a higher density of vertices will result in a more optimal solution, but will also be more complex. A better starting arrangement of vertices can curtail this exponential increase in complexity and still yield a near-optimal solution.

Known locations for threats, such as radar sites, can be used to build the graph. Since threats and radar are generally to be avoided, a graphical approach based on Delaunay Triangles and their geometric dual, Voronoi diagrams, arranges the vertices in a much more natural layout[8]. McLain[9, 10] and Beard[10] developed a Voronoi-based approach for UAV trajectory generation. Figure 2.2 illustrates a typical Voronoi diagram.

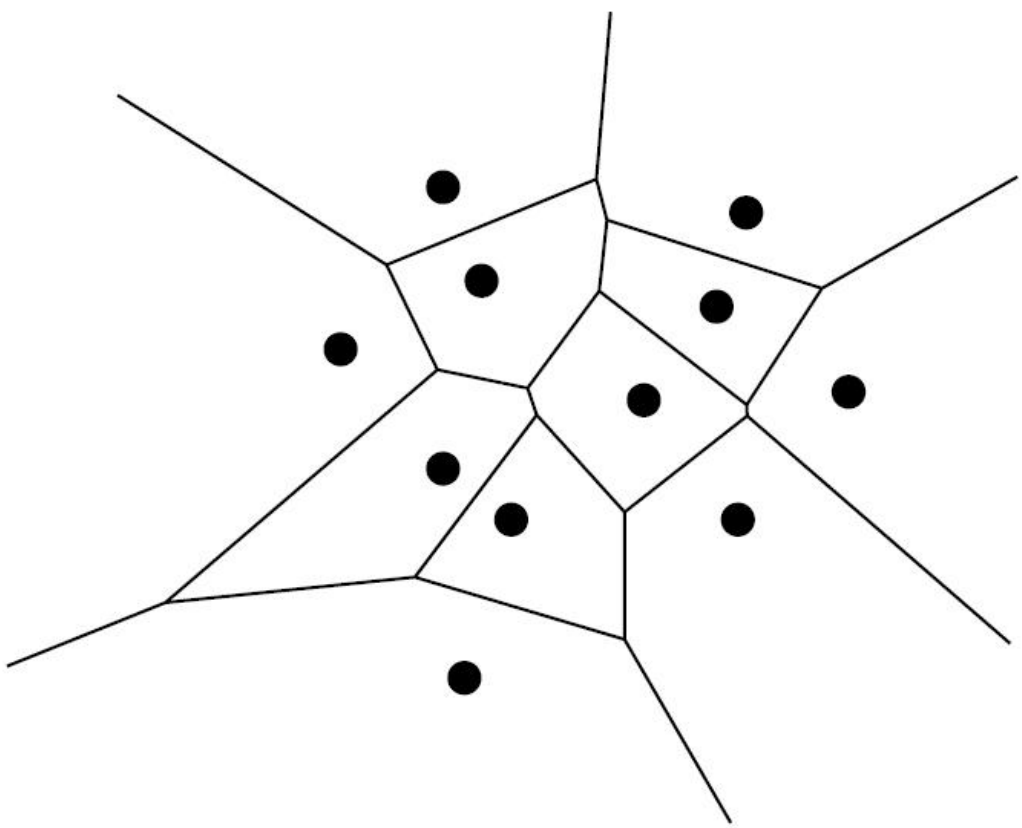

Figure 2.2: Voronoi diagram for threat locations (shown as black dots)

A Voronoi diagram places vertices such that the edges connecting any two will be equidistant from the two closest sites (in this case, threats or radar sites). The diagram is constructed without regard to starting or finishing points, and thus these must be added into the graph. In McLain and Beard's approach, the starting and finishing points are connected to the three closest vertices.

Once the Voronoi diagram is complete, costs are assigned to each of the edges. The general approach is to construct costs based on fuel costs and threat costs. When costs are assigned, the Voronoi diagram is searched to determine the lowest cost path from the starting position to the finishing position. A number of algorithms can be used for this – McLain and Beard use Dijkstra's algorithm[11], but Eppstein's *k-shortest paths* algorithm can also be used[9, 12]. For a graph with V vertices and E edges, the complexity of Dijkstra's algorithm is $O(V \log(V)+E)$; thus the complexity of the problem is always predictable. Once a solution is generated, it will be the lowest cost path for a UAV from a given starting position to a known finishing position. It may neither be the shortest path, nor the safest path, but will be the lowest in cost according to whatever cost function was assigned. Voronoi can also be modified if certain sites are weighted (such as flying between a powerful radar and a weak one), resulting in curves known as circles of Apollonius[13].

For graph-based path planning, the resulting path must be made flyable for the aircraft. There are several techniques for accomplishing this goal. The first involves discretization of the path. This 'chain path' is made flyable by smoothing[9]. Another method involves overlying splines to the path, as demonstrated by Judd and McLain[14]. The Voronoi diagram and other graph-based methods have advantages that the optimal solution from the graph is always found and that the complexity of the solution is always bounded. Thus, the problem can be setup such that it can achieve real-time performance.

The second approach to UAV path planning is classical optimal control. This approach, using Calculus of Variations, had been used since the 1960's for aircraft path

planning. In it, a cost function consisting of a path length cost, a proposed 'radar cost', and a turning cost are subject to constraints of the starting and final aircraft states and a simple model of the aircraft kinematics[8, 15, 16]. The dynamic constraints assure that the final path will be flyable. Although optimal control produces an optimal solution, computation complexity means that it may not be able to achieve real-time performance.

The third approach to UAV path planning is one using virtual potential fields and forces, as proposed by Bortoff [8]. In this method, a chain of masses connected to each other by springs and dampers represents a UAV path. Obstacles to be avoided, such as radar and threats, have repulsive force fields that shape the path until equilibrium is reached. This method has had the smallest amount of research performed among the three, though Bortoff concludes that the method is quite promising for a uniform radar field.

2.2 Path Planning/Task Allocation Approaches

Whenever task allocation is added to the path-planning problem, the complexity greatly increases because the task allocation and the trajectory generation are highly coupled. The cost for each UAV to visit a particular target is clearly a function of the path taken. If trajectory optimization could be performed for all the possible permutations of vehicle to target, the task assignment could be performed, and a globally optimal, dynamically feasible solution would be reached. Unfortunately, this can realistically be performed only for a very limited number of vehicles and targets. Otherwise, the number of possible permutations makes the probably computationally impossible for real-time in-flight performance.

Aside from specialized, proposed approaches such as a genetic algorithm proposed by Chen and Cruz[17], there have been three main approaches for solution of the task allocation and path-planning problem. Jonathan How and his group at MIT researched the first of these approaches. In this approach, the coupling between task allocation and path planning is partially decoupled[18]. From the known locations of no-fly

zones, threats, waypoints, and targets, the first step is the creation of polygons for threats and no-fly zones. The vertices of these polygons are then connected to polygons and to the vehicle and target using a 'line-of-sight' approach. Once all possible graph segments using the polygons and line-of-sight are formed, the Floyd-Warshall All-Shortest Path algorithm[18] is employed to find the shortest paths (where cost is based solely on path length) for all vehicles to all targets and waypoints. Once these paths are known, the basic task allocation problem is formulated as a *Multi-dimensional, Multiple-Choice Knapsack Problem* (MMKP) [19]. In this type of knapsack problem, one element must be chosen from each of the multiple sets. Each choice yields a benefit but uses up a resource dimension. The goal of the MMKP applied to this problem is to minimize overall cost while selecting a single path for each vehicle and being constrained to ensure that each target and waypoint is visited. Once task assignment has been performed, a more refined trajectory generation scheme is used to make the chosen paths flyable. If the flyable paths are sufficiently different from the original paths used to calculate the task allocation, the problem can be resolved using different, more refined, paths to begin with. To cope with a dynamic environment, How proposes using a local repair method [18] for reshaping an individual UAV's path or a sub-team reallocation for those UAVs directly affected by a change in environment.

The second approach for solution of the path planning and task allocation problem has also been researched by How and his group at MIT[20]. This method combines the task allocation and trajectory planning into a single *Mixed-Integer Linear Program* (MILP) optimization problem[20, 21, 22]. In order to create a linear (as opposed to nonlinear) program, the aircraft dynamics are linearized. These dynamic constraints, plus other constraints such as each UAV only having one selected path and each pre-assigned target and waypoint being visited, create the variables for the MILP problem. This method is guaranteed to find the globally optimal solution that provides detailed trajectories for each vehicle to reach its allocated waypoints in minimum time, but it is computationally intensive. Although experiments involving ground vehicle have been performed to demonstrate the usefulness for small-scale path planning and task allocation problems[23], a

MILP strategy is typically used for a benchmark, as it is a centralized scheme that is computationally inefficient for real-time applications[24].

The third approach is a hierarchical control scheme that has been developed by Chandler and Pachter at Wright-Patterson Air Force Base[25]. This hierarchical decomposition deals with the coupling-induced complexity and a method to reduce it[26]. There are four layers within the hierarchical autonomous controller[27, 28]. The first layer is the decision-making layer. This layer performs the task allocation function by using a market-based bidding method and also assures that all mission objectives and sub-objectives are met. The second layer is the path planning level. This layer coordinates cooperative search, classification, attack, damage assessment, and rendezvous. The third layer is the trajectory-planning layer, which the individual UAVs perform for themselves. The fourth layer is a redundancy management layer, which ensures accurate following of desired trajectories. Whenever task allocation is needed, each of the vehicles performs trajectory planning in their third layer. The top, centralized layer uses an auction, such as a forward Gauss-Seidel auction, a forward Jacobi auction, or a forward/reverse auction to perform the task allocation[29]. The auction resembles a stock exchange, in that each vehicle 'bids' on an assignment. Vehicles with a higher bid (meaning higher cost to perform the assignment) trade off with vehicles that have a lower cost to perform the assignment. The goal of the auction is to minimize the overall cost of performing all assignments. There has been much research performed using this approach [25-33], and currently the U.S. Air Force's LOCAAS UAV (discussed in Section 1.2) uses this scheme.

Of the three methods, the first method by MIT and the third method by WPAFB have been shown to be the most appropriate for path planning and task allocation performed aboard actual UAVs. While the results of both methods are suboptimal, research performed has shown that they perform well, without the complexity associated with an optimal solution as found using the second method.

2.3 Decentralized Control and Communications

The first and third methods mentioned in the previous section have been shown to be more appropriate for actual implementation in part due to the decoupling of the tasks. Especially with the third method as researched by the Air Force, individual UAVs make calculations for themselves in the decentralized portion of the scheme. The topmost layer of the scheme then uses these calculations for task allocation[34]. In How's research for the first method, he proposes distributing the optimization of the selected paths to the individual UAVs. These methods are partially decentralized, meaning that there is still some 'supervisory' centralized processor[35-37] that makes group decisions. For both the centralized and decentralized schemes, communication among UAVs is an issue. For a centralized scheme, delay or loss of communication means that the vehicles will not receive any instructions for performing tasks, whereas in a decentralized scheme, each vehicle can still perform tasks, though there may be some repetition of tasks and loss of others. Mitchell, Schumacher, and Chandler studied the effects of a delay using the hierarchical control methods[38]. Communication delays of 1 to 3 seconds were simulated and resulted in a significantly decrease in successful attack and verification, though tasks were still completed. A delay or loss of communication implies a lack of cooperation, but for UAVs that are involved in the decision-making process, tasks can still be performed[39].

Chapter 3
Development of the Path Planning/Task Allocation Scheme

3.1 Discussion of Setup

In selection of methods for performing path planning and task allocation, the type of mission envisioned is crucial. For the research presented here, the problem statement given in Section 1.3 dictates the following:

> *Given 'nuav' UAVs with 'nzones' no-fly zones such as mountains or political boundaries, and given 'ntarg' targets or waypoints to visit, the UAVs must accomplish a mission such as visiting each target or waypoint while minimizing an overall cost to the group. Extending this basic formulation to add realistic constrains and boundary conditions include timing constraints, such as a preliminary target needing to be reconnoitered prior to attacking. Also dynamic constraints on planned paths, such as maximum linear velocities for UAVs and maximum angular rates for rolling performance need to be accounted for. Furthermore, the problem may be time varying, where there are addition/removal or targets, loss of UAVs in the team, and loss of communications may occur. Also, in the role of high-risk but high-value missions, there will also be 'nthreats' threats in the scenario that the UAVs should avoid.*

This setup is considered to be appropriate for the mission of high-risk by high-value target attack. In this mission, the high-valued targets are known, the area having been reconnoitered previously by possibly other UAVs or even satellite intelligence. During this reconnaissance, threat and no-fly zone information is also given. The mission must still be able to account for a dynamic environment where new targets may appear, known targets may disappear, and real threats can 'pop-up' and destroy UAVs working in a team.

The literary review of Chapter 2 presented three main approaches for the solution of the path planning and task allocation problem. As concluded, the use of a *Mixed-Integer Linear Program* based approach is only appropriate for a benchmark. Of the remaining two, for a high-risk by high-value mission, the approach presented by How will be seen as more suitable. Currently, the hierarchical control scheme is quite suitable for a highly dynamic environment that a flying munition such as LOCAAS is expected to encounter. These UAVs perform the Suppression of Enemy Air Defenses role by being released in an area thought to contain some threats and enemy air defenses. As the UAVs search for targets (which are air defenses in the SEAD mission, so there are no threats), any changes in the environment cause the market-based bidding scheme to be employed.

While highly effective for such missions, whenever known target locations and no-fly zone and threat-avoidance are considered, a method similar to How's approach is more desirable. With this type of approach, all the a priori information about the targets, threats, and no-fly zones can be considered during path planning and task allocation, while certainly being adaptable to dynamic environment changes. The first part of this research presents a path planning and task allocation approach that shares similarities with the one presented by How et.al. in "Co-ordination and Control of Multiple UAVs". The presented approach uses a *Multi-dimensional, Multiple-Choice Knapsack Problem* algorithm for solution of the task allocation portion, as does How's approach, but the steps leading to the MMKP employment are quite distinct. The information used to set up the approach presented here includes the following:

- Information about UAV positions, altitude, velocity, and heading angle;
- Information about target positions, deemed target values, and the current state of the target (whether it is confirmed as a target, reconned, attacked, or battle-damage assessment performed)
- Information about threat positions, effective ranges, and probability of kill
- Information about no-fly zone positions and size

3.2 Voronoi Diagram Generation

The first step in this approach is the determination of possible paths that the UAVs could take to reach targets. Several methods were discussed in the literary review, including graph-based methods, optimal control, virtual potential fields, and the line-of-sight method described in How's method. Of these, the graphical methods have the advantage. Optimal control tends to be computationally inefficient, and the virtual potential field method is largely unresearched. While the line-of-sight method theoretically finds the shortest paths to initially choose from, the threats must be modeled the same as the no-fly zones, with definitive boundaries and vertices surrounding. This is less than optimal with threats because the probability of being destroyed if the UAV enters the range of the threat is not considered. Though the UAV would incur an additional cost due to the possibility of being destroyed, this may be desirable, as the overall path may be cheaper from the lowering of the distance cost. The inability to pass within the boundaries of a threat also causes a certain dilemma when considering that multiple threat ranges can overlap, and targets can possibly (an most probably will) be inside of the effective range of one or many threats.

Graphical methods do not take into consideration the boundaries of no-fly zones or threats. These methods must account for these boundaries with additional costs such as a probability of being destroyed cost for entering the effective range of a threat and an infinite cost for flying into the boundary of a no-fly zone (more on the cost function in the following section). Of the possible graphical methods, Voronoi diagrams were concluded to have many advantages for path planning and have been used in this research approach.

In order to properly define a Voronoi diagram, the Euclidean distance between two points *p* and *q* must be defined for points in a plane:

$$dist(p,q) \equiv \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2} \tag{3.1}$$

The sites for the Voronoi diagram are defined as:

$$p \equiv (p_1, p_2, \ldots, p_n) \tag{3.2}$$

which are a set of *n* distinct points. The Voronoi diagram of these sites is defined as the subdivision of the plane into *n* cells, one for each site, with the property that a point *q* lies in the cell corresponding to a site p_i if and only if the distance $dist(p,q_i)$ is less than the distance $dist(p_j,q)$ for each p_j in *p* where *i* is not equal to *j*[40]. Each site *p* corresponds to a single Voronoi cell, which is the intersection of a number of half-planes. The Voronoi diagram is a planar subdivision whose edges are a number of straight-line segments. Figure 3.1 illustrates a typical Voronoi diagram showing 13 no-fly zones, represented by black dots, and 12 threats, represented by green circles. The UAV positions are shown in the lower left corner while the target positions are shown in the upper right.

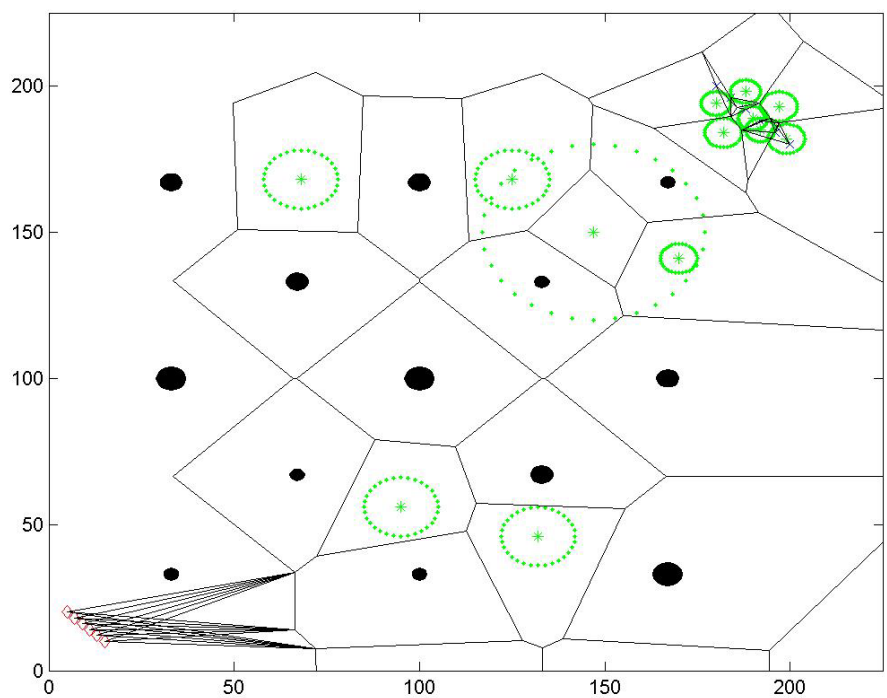

Figure 3.1: Voronoi diagram with 25 sites

An algorithm for computing a Voronoi diagram is illustrated next. This algorithm is a plane sweep algorithm commonly known as *Fortune's algorithm*, which is shown in *Computational Geometry: Algorithms and Applications*[40].

Algorithm for computation of Voronoi Diagram

Input. A set of point sites in the plane
Output. The Voronoi diagram given inside a bounding box in a doubly connected edge list structure

1. Initialize the event queue Q with all site events.
2. **while** Q is not empty
3. **do** Consider the event with largest y-coordinate in Q .
4. **if** the event is a site event, occurring at site p_i
5. **then** HANDLESITEEVENTp_i
6. **else** HANDLECIRCLEEVENTp, where p is the lowest point of the circle causing the event
7. Remove the event from Q .
8. The internal nodes still present in T correspond to the half-infinite edges of the Voronoi diagram. Compute a bounding box that contains all vertices of the Voronoi diagram in its interior, and attach the half-infinite edges to the bounding box by updating the doubly-connected edge list appropriately.
9. Traverse the half-edges of the doubly connected edge list to add the cell records and the pointers to and from them.

The procedures to handle the events are defined as follows.

HANDLESITEEVENT(p_i)

1. Search in T for the arc $\langle$ vertically above p_i, and delete all circle events involving $\langle$ from Q .
2. Replace the leaf of T that represents $\langle$ with a subtree having three leaves. The middle leaf stores the new site p_i and the other two leaves store the site p_j that was originally stored with $\langle$. Store the tuples $p_j\ p_i$ and $p_i p_j$ representing the new breakpoints at the two new internal nodes. Perform rebalancing operations on T if necessary.
3. Create new records in the Voronoi diagram structure for the two halfedges separating $V\ p_i$ and $V\ p_j$, which will be traced out by the two new breakpoints.
4. Check the triples of consecutive arcs involving one of the three new arcs. Insert the corresponding circle event only if the circle intersects the sweep line and the circle event isn't present yet in Q .

HANDLECIRCLEEVENT(p)

1. Search in T for the arc $\langle$ vertically above p that is about to disappear, and delete all circle events that involve $\langle$ from Q.
2. Delete the leaf that represents $\langle$ from T. Update the tuples representing the breakpoints at the internal nodes. Perform rebalancing operations on T if necessary.
3. Add the center of the circle causing the event as a vertex record in the Voronoi diagram structure and create two half-edge records corresponding to the new breakpoint of the Voronoi diagram. Set the pointers between them appropriately.
4. Check the new triples of consecutive arcs that arise because of the disappearance of $\langle$. Insert the corresponding circle event into Q only if the circle intersects the sweep line and the circle event isn't present yet in Q.

This algorithm is implemented in the MATLAB function as found in *voronoi.m*, which is shown in Appendix A.

The number of vertices in the Voronoi diagram of a set of *n* point sites is at most $2n-5$ and the number of edges is at most $3n-6$ [40]. From this theorem it is seen that for an insufficient number of sites (threats and no-fly zones in this case), the Voronoi diagram will either not be able to be computed or will have a small number of edges for finding appropriate paths. To work around this difficulty, 16 extra sites are added around the edges of the known battlefield. This ensures that even without any threats or targets, there will be edges to choose paths from. Once this is accomplished, the next step is to compute the Voronoi diagram, which as mentioned before is *voronoi.m*.

The computation of the Voronoi diagram is the first major step in this path planning and task allocation research. The MATLAB code implementing this is *vrn_diag_gen.m*, which is shown in Appendix A. After the computation of the Voronoi diagram, the UAV locations and the target locations must be added into its list of vertices. For each of the locations of UAVs and targets, the 3 closest vertices are found. Three edges between these vertices and the location are formed and added to the edges of the Voronoi diagram. This completes the Voronoi diagram section of the approach, and next follows the cost assignment and determination of the cheapest paths for each permutation.

3.3 Dijkstra's Algorithm and Cost Assignment

Once the Voronoi diagram is complete and the UAV positions and target positions are connected, a path planning method must determine the optimal path for each permutation of UAV to target. This consists of two separate parts – first, a cost function must be developed and applied to each edge of the Voronoi diagram, and second, the edges must be searched to determine the optimal path, which is defined as the combination of graph edges that connects the UAV to the target with the lowest possible cost.

The first task in this section of the approach is the assignment of costs to each graph edge. The cost function developed here consists of three separate parts. The first part of the cost relates to the fuel cost. Since typically UAVs will be flying at a constant speed, the fuel required to fly along an edge will be proportional to the length of the edge. Thus, the first part of the cost function is a distance cost. The second part of the cost is that which is related to no-fly zone cost. No-fly zones could be possibly mountains or political boundaries. Offensive UAVs crossing a political boundary could certainly be disastrous and should never be allowed. Similarly, a UAV crossing a physical boundary (crashing into a mountain) is also unacceptable. Thus, to ensure that crossing political and physical boundaries is never a cheapest path, a cost of infinity is assigned to each edge that intersects such a boundary. The last part of the proposed cost function is associated with threats. A typical threat can be visualized as a munition (whether anti-aircraft artillery or surface-to-air missile) that has an effective range which inside has a 'probability-of-kill' for destruction of intended aircraft. Table 3.1 illustrates some typical threats and their associated effective ranges and probabilities-of-kill.

Table 3.1: Typical threats[41]

Name	KS-19	SA-7 Grail	Crotale	SA-2
Type	100mm - Antiaircraft Artillery	Man-Portable SAM	SAM	SAM
Effective range	4000 meters	5000 meters	10,000 meters	30,000 meters
Probability of kill	40%	50%	80%	80%

These threats are used as examples of real-world threats that might be encountered in current conflicts. These particular threats were compiled by selection of several arms available to the former Iraqi regime. Figure 3.2 depicts a launched Crotale "Rattlesnake" SAM that can be used effectively inside at 10-kilometer range, with a probability of intended aircraft destruction of 80%.

Figure 3.2: Crotale "Rattlesnake" surface-to-air missile

Thus, the cost assigned due to threat boundary intersection is as follows: for each permutation of edges and threats, the length of edge is found, and the Euclidean distances of the first (starting) vertex of the edge to the center of the threat and the second (finishing) vertex to the center of the threat are found. These distances are provided in the following equations:

$$Edge\_length = \sqrt{\left(v_{s,x} - v_{f,x}\right)^2 + \left(v_{s,y} - v_{f,y}\right)^2} \quad \textbf{(3.3)}$$

$$V_{start}\_to\_center = \sqrt{\left(v_{s,x} - c_x\right)^2 + \left(v_{s,y} - c_y\right)^2} \quad \textbf{(3.4)}$$

$$V_{finish}\_to\_center = \sqrt{\left(v_{f,x} - c_x\right)^2 + \left(v_{f,y} - c_y\right)^2} \quad \textbf{(3.5)}$$

Next, the 3 distances are used in the following equation to find the distance from the starting vertex to the point where the perpendicular of the edge to the center of the threat intersects the edge.

$$V_s\_to\_intersect = \frac{\left(Edge\_length^2 + V_{start}\_to\_center^2 - V_{finish}\_to\_center^2\right)}{2 * Edge\_length} \quad \textbf{(3.6)}$$

If this distance from the starting vertex to the intersection is greater than zero (meaning it is past the starting vertex in the direction of the other end of the edge) and is less than the length of the edge, then the closest point on the edge to the threat is that point of intersection. Equation 3.7 gives that distance.

$$Closest\_distance = \sqrt{V_s\_to\_center^2 - V_s\_to\_intersect^2} \quad \textbf{(3.7)}$$

If the distance from the starting vertex to the intersection is negative, the closest point on the edge is the starting vertex. Otherwise, the distance is greater than the length of the edge, and the closest point is the finishing vertex.

Once the closest point on the edge is computed, the effective range of the threat and the distance between that edge and the center of the threat are compared. If the edge falls within the range of the threat, a threat cost is added to the distance cost of the edge, shown by Equation 3.8.

$$Edge\_cost = W_1 * Edge\_length + W_2 * Threat\_prob\_of\_kill \quad \textbf{(3.8)}$$

In this equation, W_1 is a weight for the cost of distance due to the proportionality of fuel to distance and W_2 is a weight for the probability of being destroyed. The preceding algorithm is implemented in the code *c_assign.m*, which again is found in the first appendix.

At this point, all edges now have realistic costs associated with flying along that edge. The next step is searching of these edges to determine the cheapest paths for each UAV to target permutation. As the section title suggests, this has been accomplished using Dijkstra's algorithm. Dijkstra's algorithm solves the cheapest path problem for a directed graph that has nonnegative edge costs[42]. The necessary inputs for the algorithm include the set of vertices and the set of ordered pair representing the edges connecting those vertices. Notice that Dijkstra's algorithm requires a graph with directed edges. This means that each edge must be designated with a starting vertex and a finishing

vertex (unlike in the threat cost assignment where the starting and finishing vertex labels are arbitrary). To overcome this difficulty, the Voronoi diagram is overlaid with two edges connecting each set of vertices. The first edge has an arbitrary labeling of starting and finishing vertices while the second, identical edge has the opposite labeling. The coding labeled *set_thc.m* (meaning tail-head-cost) solves this. This code first renames all vertices with integers from 1 to *n* and refers to them in this manner instead of using their coordinates. The ordered pairs of vertices representing directed edges and their associated cost form an adjacency list.

For implementation of Dijkstra's algorithm, a weighted adjacency matrix must first be formed. A weighted adjacency matrix is defined as a square *n*-by-*n* matrix whose entry in row *i*n and column *j* indicates the cost from the *ith* to the *jth* vertex[43]. Figure 3.3 shows an example of a directed graph with costs.

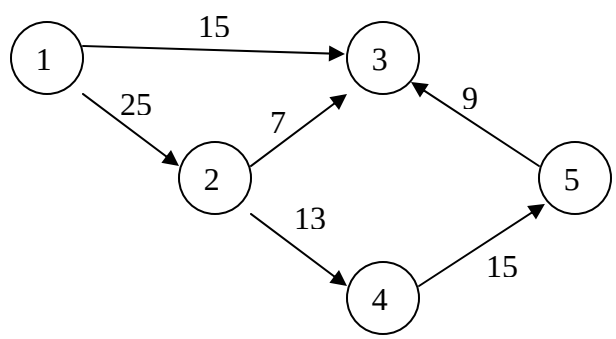

Figure 3.3: Example directed graph with costs

The corresponding weighted adjacency matrix for this figure is:

$$\begin{bmatrix} 0 & 25 & 15 & 0 & 0 \\ 0 & 0 & 7 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 \\ 0 & 0 & 9 & 0 & 0 \end{bmatrix}$$

The adjacency matrix is formed using the file *list2adj.m*. This file is available from the MATLAB toolbox Matlog[44].

The algorithm for Dijkstra with inputs of the adjacency matrix and the beginning vertex (a UAV position) and finishing vertex (a target position) works by constructing a subgraph *S* such that the cost of any vertex *v* in *S* from the beginning vertex s is known to be minimum[44]. The algorithm[43] is as follows:

1. **for** each vertex *v*, set d(v), the cost of reaching that vertex, to infinity
2. Set d(s), the cost of reaching the current vertex from itself, to zero
3. Initialize *S* a an empty set
4. Initialize *Q* as a set of all the vertices
5. **while** *Q* still has vertices in it,
    a. **find** vertex *u* in *Q* that has the lowest *d(v)* value
    b. include the vertex *u* in the set *S*
        i. **for** each vertex *v* with is connected to *u* with an edge
            1. **if** *d(v) > d(u) + edge cost*
            2. **then** *d(v) = d(u) + edge cost*
    c. remove vertex *u* from *Q*

This algorithm continues until the cheapest cost from the UAV position to the target position is found. This algorithm is implemented in the Matlog toolbox function *dijk.m*. The function outputs the total cost for the individual UAV to reach a target, and the order of vertices the path takes. This concludes the selection of the cheapest paths for each UAV to target permutation.

3.4 Path Shortening and Flyability

As mentioned previously, developing paths based on a Voronoi diagram has limitations for battlefields with smaller numbers of sites (the threats and no-fly zones). To address this issue, it was suggested that additional sites should be added into the list of

sites, ensuring that Voronoi produces acceptable possible paths. However, this adds an unwanted side effect. When the cheapest paths are selected, some of the paths may have unnecessary 'kinks' due to Voronoi avoidance of these sites that do not represent either threats or no-fly zones. This issue can be dealt with by using a path shortening method based on line-of-site. Whenever the method of line-of-sight path shortening is employed at this point, the best features of Voronoi diagrams are coupled with the best features of line-of-sight path generation. The previous disadvantages of the line-of-sight method were highlighted as the lack of realistic threat modeling and the situations where threats overlapped each other or desired targets. The modified line-of-sight version presented here removes these disadvantages.

The file *path_shrtng.m* uses the methods discussed in this section. Adding a number of new vertices along each edge modifies the previously selected cheapest paths. The number of new vertices is variable, but typically ten new vertices are added per edge. These vertices take the place of the vertices surrounding threats and no-fly zones are proposed previously for a line-of-sight method. Once these vertices are added, new edges are effectively created. With these new edges, the modified line-of-sight method can be implemented.

Since UAV paths already selected from the above sections may include passing into threat boundaries, the modified line-of-sight approach must address this. The first step the approach takes is identifying which UAV pass though which threats and at what range. The next step is to essentially decrease the range of the threats for these UAVs. These vehicles have already incurred a threat cost, thus that UAV is permitted to enter that threat up to the radius it had previously before. Each UAV may 'see' a different set of threats at this point, representing where its previous path went. It should be noted, however, that for each UAV to target permutation, all of the threats that it did not enter as part of its previously selected path remain unmodified. The only boundaries that are reduced are the ones that the individual UAV passes through.

The path-shortening algorithm executes for each UAV to target permutation. This algorithm begins by selecting the UAV position for a single permutation. This position becomes the starting vertex in the list of vertices that produce the path. From the starting vertex, the algorithm couples that vertex with the target vertex and checks the produced edge to see if it intersects a threat or no-fly zone via the method discussed in the previous section. If the edge is found to intersect a boundary line, the starting position is coupled with the vertex immediately preceding the target position. The algorithm continues to choose vertices successively backward until a combination that produces no intersections with any boundary is found. The vertex at the end of this edge becomes the new second vertex of the path and the new starting vertex for the algorithm to pair up with the target position. The algorithm continues until the target position is reached, which can occur in as few as a single edge from the UAV position to the target position to as many as the number of edges selected from the original Voronoi diagram.

The next issue to address is the flyability issue. In Section 2.1, two methods were presented for this task. The first was one that discretized the paths into chains and used smoothing effect via forces. The second method was one in which splines were used. The spline approach was considered to be excellent for producing flyable paths. However, upon implementation, it was soon to found to be too computationally intensive. A new method is presented here to solve this problem.

Fillets can be added to intersection of edges in order to make paths more flyable. As for aircraft dynamics, the concept being addressed deals solely with a minimum turning radius. Though a full review of aircraft dynamics is covered in a subsequent chapter, the concept of minimum turning radius for an aircraft is the tightest turn that the aircraft is physically able to make. This property is dependent upon several variables, including the aircraft inertia properties and velocity. For a known minimum turning radius, each intersection of edges for the paths can be filleted to account for simple aircraft dynamics.

This concept is found using several equations and a few trigonometric relations. Adding fillets begins with selecting the first three vertices of a path. These three vertices will form some sort of angle that the aircraft will by some degree not be able to completely follow. These vertices are labeled *Start*, *Middle*, and *Finish*, relating to their position in the path. The first calculations needed are the Euclidean distances from the *Start* to the *Middle* vertices, from the *Middle* to the *Finish* vertices, and from the *Start* to the *Finish* vertices. These distances are labeled *SM*, *MF*, and *SF*, respectively. The angle formed by the intersection of the two edges is called α, and can be found using the following equation, which is simply the law of cosines:

$$\alpha = \arccos\left(\frac{SM^2 + MF^2 - SF^2}{2 * SM * MF}\right) \tag{3.9}$$

Thus, the lengths *SM*, *MF*, and *SF*, and the angle α are now known. A circle of minimum turning radius is now fitted to the angle caused by the intersection of the edges. The circle is fitted such that each edge forms a tangent on the circle. The place where the edge touches the circle is where a new vertex should be placed. From the *Start* position traveling along the path, it can be seen that upon reaching the position of the first new vertex, the vehicle should follow the circle until it reaches the next vertex, upon which it the follows the original path on toward the *Finish* vertex.

The position of the new vertex can be found by noting that a line of the minimum turning radius length connecting the center of the circle to the tangent intersection of the circle and the edge *SM* is obviously perpendicular to the edge. The radius is known, a right angle is found, and the angle formed between the edge and a line connecting the *Middle* vertex and the center of the circle is half of α. This leads to Equation 3.10 that defines the length entitled *Fillet*.

$$Fillet = \frac{Min\_turn\_radius}{\tan\left(\frac{\alpha}{2}\right)} \tag{3.10}$$

The following figure illustrates the filleting principle. The circle meets both edges on a tangent, and the new vertices are found using the length *Fillet*, as shown in the figure.

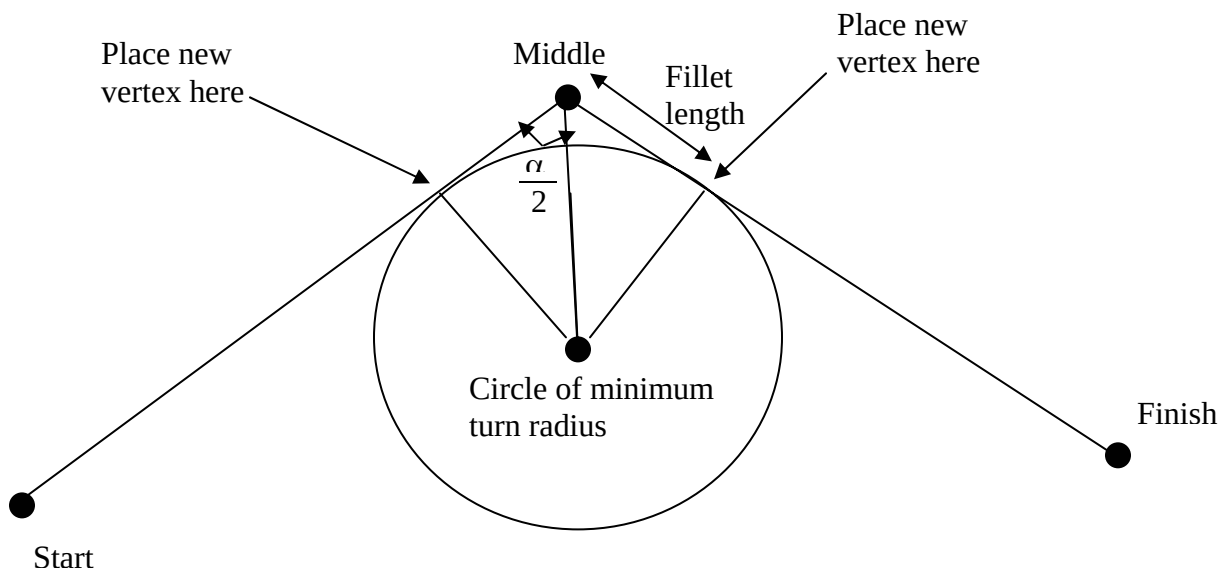

Figure 3.4: Picture illustrating fillet principle

This procedure creates the two new fillets and removes the vertex *Middle*. This is continued by moving along the path and re-labeling new vertices with *Start*, *Middle*, and *Finish* until the target vertex is labeled *Finish*, at which point the path can be considered flyable. Each path representing every permutation of UAV to target is made flyable in this manner.

A second task for flyability is met when considering that a current path is not formed with respect to the aircraft's heading angle. Though the path is considered to be a flyable one, this can only be if the UAV was initially facing directly towards the first vertex along the path from its initial starting vertex. This will only occur a small percentage of the time, so the path must be supplemented at the beginning with several segments that get the UAV onto the path facing the correct direction.

As the location of the next vertex is not guaranteed to be any specific distance away from the starting vertex, it is unacceptable to simply let the aircraft attempt to turn in order to align itself with the path aside from relatively small angular differences. Depending on how close the UAV is to the next vertex and how important reaching that vertex is, a vehicle could potentially overshoot its intended target. A method is devised here that adds the minimum length section to the beginning of the path and allows the UAV to turn as quickly as possible to arrive on the selected path starting from the same initial vertex but now facing with the correct heading angle.

This methods shares similarities with the theory behind the fillets presented in the preceding pages and is much an extension of it. For an aircraft traveling along a given heading angle and suddenly re-planned and assigned a new path with a different heading angle, the quickest method to get on the new path with the correct heading angle without the possibility of overshooting any target will be to fit two circles of minimum turning radius to the old and new paths, with each circle being tangent to one of the paths and both circles being tangent to each other. To illustrate this concept, Figure 3.5 shows two different paths. This plot begins with a UAV initially with a heading angle of –90 degrees (heading toward the bottom of the plot). The new path assigned to it has a heading angle of 0 degrees (heading toward the right edge of the plot). Whenever the new path is assigned, the UAV is located in the center, where the two paths cross. In order to get on the new path with a minimum amount of time, the aircraft will begin by flying along the current path heading at –90 degrees. Upon reaching the tangent with the lower left circle (which has a radius equal to the aircraft's minimum turning radius), the UAV will begin following the circle. At the tangent between the two circles, the aircraft will follow the second circle of minimum turn radius for the short distance until it reaches its initial start point. The aircraft will now be heading exactly 0 degrees, toward the right of the plot, starting exactly from where the new path was planned to start.

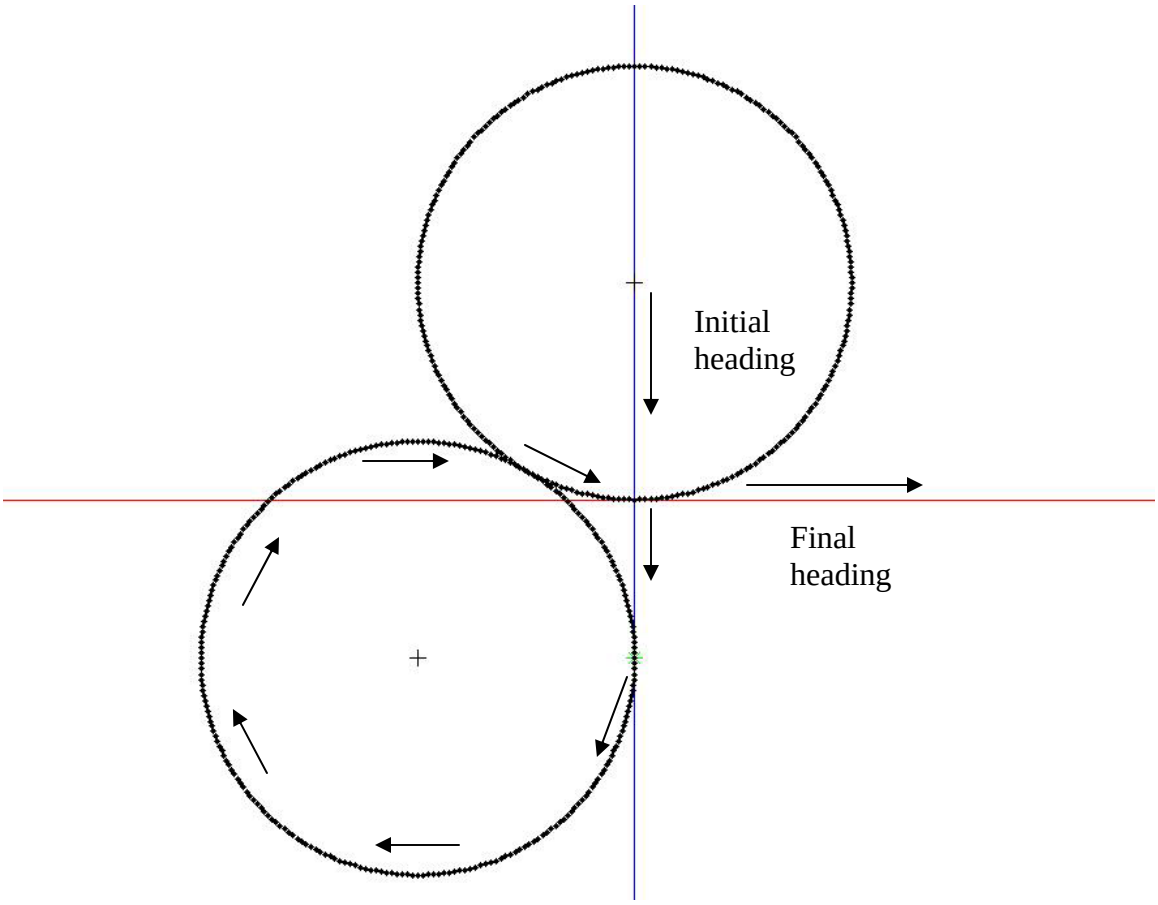

Figure 3.5: Example of heading angle solution

This method can be used for any change in heading angle. The next example demonstrates the effects of having a new path such that the heading angle flips, and the aircraft must turn around. Once again, the vehicle begins by continuing along its current path until it reaches the tangent of the first circle with the current path. It follows this circle until it reaches the tangent of the two circles, where is beings to follow the other circle. Upon reaching its initial location, where the second circle is tangent to the new path, the aircraft follows the newly assigned path now currently heading in the correct direction to accurately follow the new path.

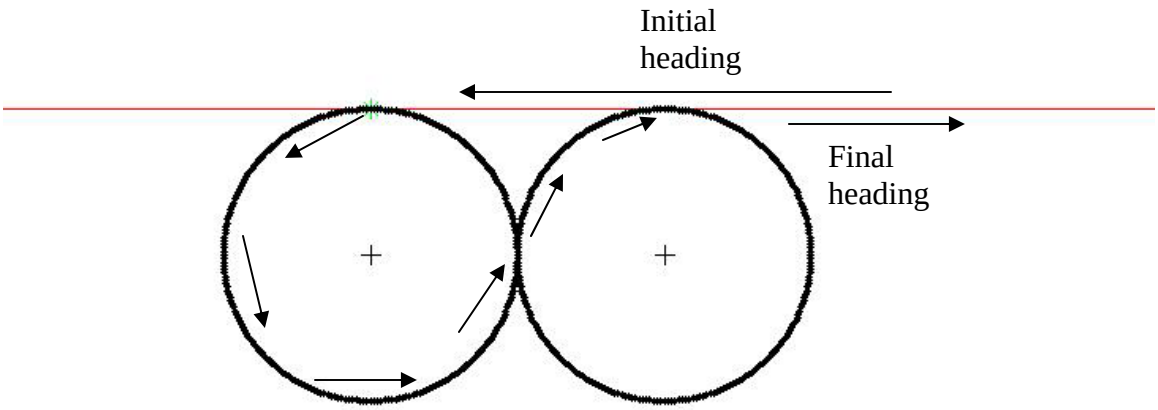

Figure 3.6: Second example of heading angle solution

Figure 3.7 is the last example meant to illustrate how this approach handles varying heading angles. In this example, the UAV is initially heading at –20 degrees and is assigned a heading angle of 25 degrees.

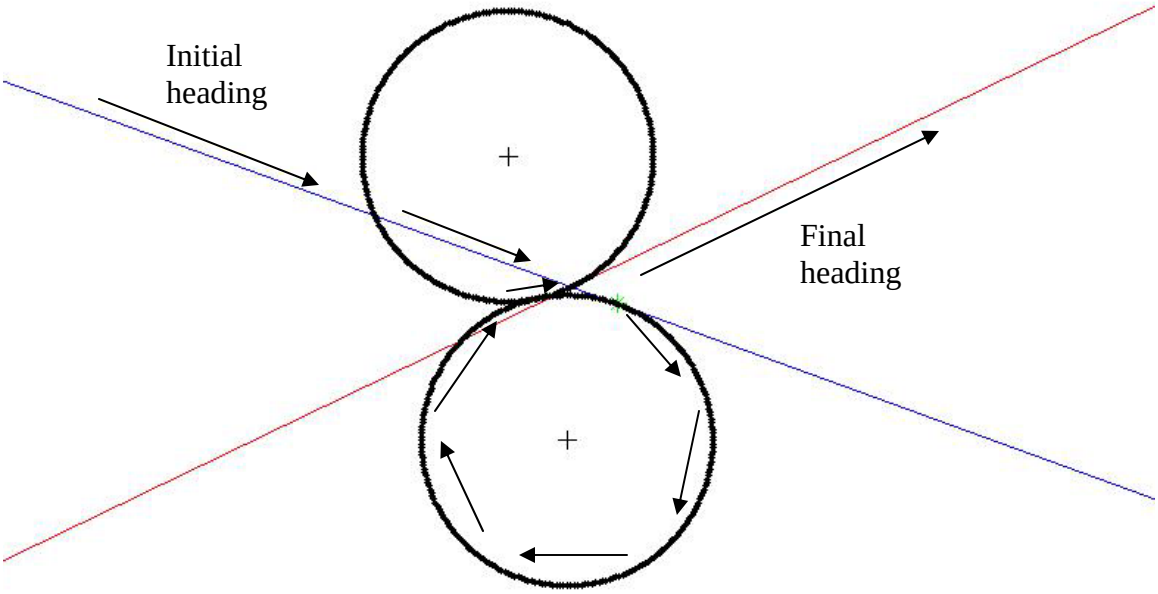

Figure 3.7: Final example of heading angle solution

This last example is getting nearing a limit that should be imposed on the usefulness of this approach. For angles with less than about 30 degrees difference, the aircraft can

follow the new path with sufficient accuracy. It should be noted that a filleting type approach could not be used here since the aircraft is already to the intersection of the two edges before the new path is assigned and corrective measures are taken.

For performing this procedure, the current heading angle and the new heading angle are found. For ease in computation, these angles are then rotated such that the new heading angle is horizontal at 0 degrees, and the current heading angle of the aircraft is rotated by the same amount. Again, for small angles of roughly 30 degrees or less difference, this procedure is omitted. The first calculation involves finding the distance the aircraft must fly before beginning to turn onto the first circle.

$$\begin{aligned} init\_dist = {} & C_1\left(\frac{|Heading\_angle|}{pi}(2min\_turn)\right)^3 \\ & + C_2\left(\frac{|Heading\_angle|}{pi}(2min\_turn)\right)^2 \\ & + C_3\left(\frac{|Heading\_angle|}{pi}(2min\_turn)\right) \end{aligned} \tag{3.11}$$

The coefficients have been determined by numerical methods for use in the MATLAB code *heading_angle_paths.m*. The coordinates of this first break point may now be calculated using the initial position of the aircraft and the distance determined from Equation 3.11.

$$x\_break = x\_uav + init\_dist * \cos(Heading\_angle) \tag{3.12}$$

$$y\_break = y\_uav + init\_dist * \sin(Heading\_angle) \tag{3.13}$$

With these coordinates, all the information for computing the two circles of minimum turning radius is at hand. The centers of the circles are found based on whether the original heading angle was rotated in the clockwise direction or counter clockwise direction. For positively rotated heading angles, the variable *ccw* will be set to negative one; otherwise, it will have a unitary value. Equations 3.14 and 3.15 are used to find the

center of the second circle. For finding the center of the first circle, the new heading angle is substituted for the current heading angle and the position of the UAV is used instead of the first breakaway point.

$$x\_circle = x\_break + init\_dist * COS\left(Heading\_angle - \frac{pi * ccw}{2} \right) \quad \textbf{(3.14)}$$

$$y\_circle = y\_break + init\_dist * SIN\left(Heading\_angle - \frac{pi * ccw}{2} \right) \quad \textbf{(3.15)}$$

Two more angles are needed to find the locations where the two circles become tangent and at what angle the first circle becomes tangent to the current path. The first angle is the one made by the horizon (the reason this system was first rotated) and the line connecting the breakaway point and the center of the first circle. The second angle is the one made by the horizon and the line connecting the center of the second circle to the center of the first circle. This now leads to the creation of vertices around the circles, starting first with the initial location of the UAV, followed by the first breakaway vertex, then with vertices around the first circle until the circles become tangent, then with the vertices along the second circle until the initial position once again becomes a vertex, and finally ending with the first assigned vertex of the new path. The coordinates are then rotated to reflect the change back to the unrotated system, and the new vertices are inserted into the new paths.

Since much change has occurred to the paths, with shortening, adding fillets, and possibly adding initial heading angle sections, updated costs are assigned to the paths using the same methods as first described in Section 3.3. It may seem redundant to have already assigned costs, only to later change them before they are used in task allocation. However, it is not computationally prudent to perform path shortening and flyability additions to such a large number of possible paths that Voronoi presents. The combination of using both a Voronoi diagram approach and a line-of-sight shortening offer advantages that neither can offer by themselves. Using the flyability methods presented in the preceding pages ensure that dynamically feasible paths will be chosen from without the complexities associated with a linear program or optimal control. This

concludes the entire path planning section and leads directly into the last section, the application of a *Multi-dimensional, Multiple-Choice Knapsack Problem* for solution to the task allocation problem.

3.5 Multi-dimensional, Multiple-Choice Knapsack Problem

The task allocation problem is solved via implementation of a *Multi-dimensional, Multiple-Choice Knapsack Problem* (MMKP), which is considered to be NP-hard [45] in the class of knapsack problems. For a typical knapsack problem, items for the knapsack must be picked such that a total value is maximized while adhering to resource constraints. A simple example of the classic knapsack problem is packing of cargo – the goal is to maximum the amount of cargo put aboard a ship or a truck or an aircraft, but resource constraints such as total weight and volume must be considered. The MMKP is a variant of such a problem. With MMKP, there are multiple groups of items. Each group has an assigned value but uses up certain resources. The objective of the MMKP is to select a single item from each group for maximizing the value while adhering to the resource constraints[46].

As applied to the current problem, the choice of a single item from a group represents a single permutation of UAV to target within the group of a single UAV. The constraints on the solution are that each target has to be visited, and each UAV has to be assigned a path. These constraints assure that tasks are assigned to all UAVs and that objectives of visiting targets are not missed by assigning multiple UAVs to perform the same task while neglecting to perform others. Instead of maximizing a value function, the equivalent benefit is derived when attempting to minimize a cost. Each permutation has already been assigned a cost as addressed in early sections, and thus it is the goal of the MMKP to use these costs to find the optimal combination of paths to minimize the cost of performing the entire mission for the team.

An example will clarify this concept. The MMKP knapsack problem of Figure 3.8 features 3 UAVS and 3 targets, and each block represents a possible path.

UAV 1	UAV 2	UAV 3
Path to target 1 Cost: 10	Path to target 1 Cost: 5	Path to target 1 Cost: 13
Path to target 2 Cost: 3	Path to target 2 Cost: 24	Path to target 2 Cost: 9
Path to target 3 Cost: 2	Path to target 3 Cost: 19	Path to target 3 Cost: 7

Figure 3.8: Example UAV to target MMKP setup

For this problem, there are six different permutations of the path combinations. Specifically, the list of permutations is found in Table 3.2.

Table 3.2: List of example path permutations and mission costs

UAV 1 Path Choice	*UAV 2 Path Choice*	*UAV 3 Path Choice*	*Cost of Paths*
3	*2*	1	39
3	*1*	2	16
2	*3*	1	35
2	*1*	3	15
1	*2*	3	41
1	*3*	2	38

From inspecting the combinations above, the cheapest combination of paths that satisfies the constraints of every target being visited and each UAV being assigned a task is the

combination of UAV 1 being assigned to target 2, UAV 2 being assigned to target 1, and UAV 3 being assigned to target 3. The total cost of performing the mission using this assignment of tasks is 15. Any other assignment of tasks results in an increased cost to perform the mission. It should be noted that the goal is only to minimize the total mission cost, not the individual costs for the UAVs. The can be seen where UAV 1 was not chosen to follow its cheapest path. It would have been cheaper for UAV 1 to be assigned to target 3 with a cost of only 2 instead of being assigned to target 2 with a cost of 3. However, such an assignment would have used up a resource allotted for target 3, and caused overall mission costs of either 16 or 39, depending on where UAV 2 and UAV 3 were assigned.

The algorithm for solution to the task allocation problem initializes by inputting each UAV to target permutation and associated cost in a matrix similar to the layout shown in Figure 3.8. Similar to Dijkstra's algorithm, the cost of assignment of any combination of paths is set to infinity. A permutations matrix that captures all the ways the UAV to target paths could be combined while adhering to the resource constraints is formed. These permutations are then searched to find the lowest cost combination. As lower cost combinations are found, they become the selected assignments unless an even lower cost combination is encountered. Once determined that there are no cheaper permutations of assignments, the MMKP reports the selected assignments and the cost to perform the mission. The code applying this method is titled *MMKP_task_allocation.m*.

The first two research objectives have now been fulfilled. Each UAV has a task assignment for visiting a target and a dynamically feasible path to complete that task. The coupling of the problem has been accounted for using this approach, and the last steps in the path planning and task allocation scheme are simple data conversion used for plotting purposes. All MATLAB code employing the methods discussed here are included in Appendix A, and are listed in the order in which they are run.

Chapter 4
Aircraft Dynamics

4.1 Introduction

The third research objective is the development of a simulation environment that employs the path planning and task allocation approach described in the previous chapter. This simulation uses a six degree-of-freedom aircraft model to follow the assigned paths that are generated for each UAV. Therefore, it is appropriate to first review the aircraft dynamics and equations of motion. More detailed descriptions and analyses than those presented here can be found in several references[47-49].

A single, nonlinear vector equation can be formulated to accurately model an aircraft:

$$\dot{\mathbf{x}} = \mathbf{f}\left(\mathbf{x}, \mathbf{F}_{total}(t), \mathbf{M}_{total}(t)\right) \tag{4.1}$$

In Equation 4.1, **x** is defined as the following vector of state variables:

$$\mathbf{x} = \begin{bmatrix} V & \alpha & \beta & p & q & r & \psi & \theta & \phi & x_e & y_e & H \end{bmatrix}^T \tag{4.2}$$

This state variable modeling consists of twelve state equations that can be divided into four groups. The first group of state variables, the translational velocity variables, consists of the true airspeed V, the aircraft angle-of-attack α, and the sideslip angle, β. The second group is the rotational velocities of the aircraft, with p, the angular roll rate, q, the angular pitch rate, and r, the angular yaw rate. The third group describes the aircraft attitude in terms of orientation of the body axes with respect to the vertical axes. This group includes ψ, the Euler yaw angle, θ, the Euler pitch angle, and ϕ, the Euler roll angle. The last group of variables describes the aircraft position with respect to an Earth-fixed set of axes. This group contains x_e, the aircraft x-coordinate with respect to the Earth-fixed x-axis, y_e, the aircraft y-coordinate with respect to the Earth-fixed y-axis, and z_e, the aircraft z-coordinate with respect to the Earth-fixed z-axis.

Certain assumptions should be noted for the following analysis of the aircraft equations of motion. First, the aircraft is considered to be a rigid body. Secondly, the mass of aircraft is not time-dependent – it is constant. Finally, a flat Earth assumption is used, where the curvature and rotation of the Earth are neglected.

4.2 Body Axes Modeling

The body axis system is depicted in Figure 4.1. Forces and moments acting on an aircraft are also shown and will be used in the following analysis. The body axis system originates at the center of gravity of the aircraft, as shown by the point. The x-axis is the longitudinal axis of the aircraft that extends along the nose to the tail. The y-axis is the lateral axis of the aircraft and is parallel with the wings. The z-axis is perpendicular with the x-y plane and points downward from the aircraft.

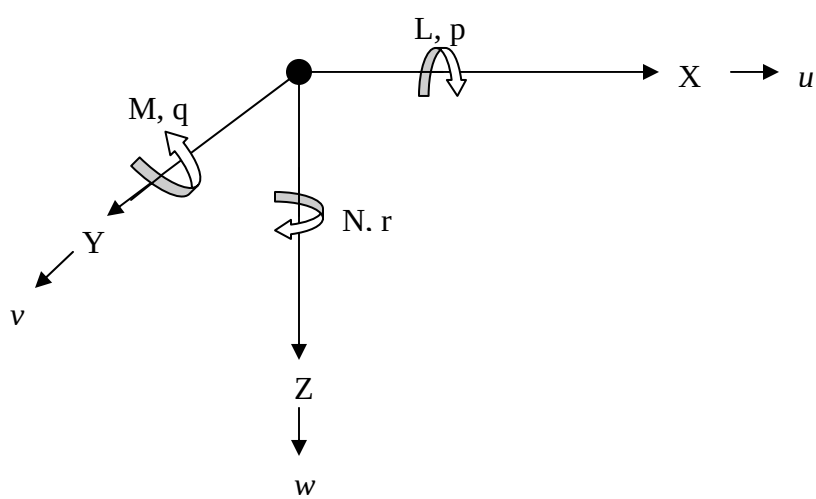

Figure 4.1: Body axis system with forces and moments

Consider a point mass δm, moving with velocity **V**, and being acted upon by force **F**. Application of Newton's Second Law yields:

$$\delta \mathbf{F} = \delta m \dot{\mathbf{V}} \tag{4.3}$$

An aircraft is considered to be a rigid body consisting of a finite number of point masses. Applying Equation 4.3 to each point mass δm and summing results in Equation 4.4.

$$\sum \delta \mathbf{F} = \sum \delta m \dot{\mathbf{V}} \tag{4.4}$$

The equation accounts for the total force acting upon the aircraft.

$$\mathbf{F} = \frac{d}{dt}\left(\sum \delta m \mathbf{V}\right) \tag{4.5}$$

where the force can be defined as:

$$\mathbf{F} = \mathbf{i}F_x + \mathbf{j}F_y + \mathbf{k}F_z \tag{4.6}$$

The center of gravity of the aircraft is defined as the average location of the weight. This location can be used to describe the velocity of the entire aircraft, using components *u*, *v*, and *w*.

$$\mathbf{V}_{c.g.} = \mathbf{i}u + \mathbf{i}v + \mathbf{k}w \tag{4.7}$$

The velocity for any point inside a rigid body is:

$$\mathbf{V} = \mathbf{V}_{c.g.} + \dot{\mathbf{r}} \tag{4.8}$$

where **r** is the vector connecting any point inside the rigid body to the center of gravity. Using this definition of velocity, Equation 4.5 becomes:

$$\mathbf{F} = \frac{d}{dt}\left(\sum \delta m\left(\mathbf{V}_{c.g.} + \dot{\mathbf{r}}\right)\right) \tag{4.9}$$

This can be divided into two separate parts,

$$\mathbf{F} = \frac{d}{dt}\left(\sum \mathbf{V}_{c.g.}\delta m\right) + \frac{d}{dt}\left(\sum \dot{\mathbf{r}}\delta m\right) \tag{4.10}$$

The second part of Equation 4.20 will be identically zero due to the definition of the center of gravity. Thus, the general force equation can be defined as:

$$\mathbf{F} = m\dot{\mathbf{V}}_{c.g.} \tag{4.11}$$

The moment developed about the center of gravity for a point mass δm located at **r** is shown in the following equation. This equation also uses the definition of angular momentum **h**.

$$\delta\mathbf{M} = \frac{d}{dt}(\mathbf{r} \times \mathbf{V})\delta m = \frac{d}{dt}\delta\mathbf{h} \tag{4.12}$$

From this, the general moment equation about the center of gravity is found to be:

$$\mathbf{M}_{c.g.} = \dot{\mathbf{h}} \tag{4.13}$$

where the moment is defined to be:

$$\mathbf{M}_{c.g.} = \mathbf{i}L + \mathbf{j}M + \mathbf{k}N \tag{4.14}$$

Next angular velocity is introduced. Angular velocity is defined as:

$$\vec{\Omega} = \mathbf{i}p + \mathbf{j}q + \mathbf{k}r \tag{4.15}$$

The angular velocity can be used to find the total velocity for any point mass according to the following equation:

$$\mathbf{V} = \mathbf{V}_{c.g.} + \Omega \times \mathbf{r} \tag{4.16}$$

The angular momentum can also be shown to be **I**, the inertia tensor, dotted with the angular velocity.

$$\mathbf{h} = \mathbf{I} \cdot \vec{\Omega} \tag{4.17}$$

The inertia tensor is given by:

$$\mathbf{I} = \begin{bmatrix} I_x & -I_{xy} & -I_{xz} \\ -I_{yx} & I_y & -I_{yz} \\ -I_{zx} & -I_{zy} & I_z \end{bmatrix} \tag{4.18}$$

Using the body reference frame described in Figure 4.1, the entire reference frame rotates with the angular velocity. The general force and moment equations given by Equations 4.11 and 4.13 then become:

$$\mathbf{F} = m\left(\frac{\partial \mathbf{V}_{c.g.}}{\partial t} + \vec{\Omega} \times \mathbf{V}_{c.g.}\right) \tag{4.19}$$

$$\mathbf{M}_{c.g.} = \frac{\partial \mathbf{I} \cdot \vec{\Omega}}{\partial t} + \vec{\Omega} \times \left(\mathbf{I} \cdot \vec{\Omega}\right) \tag{4.20}$$

The force equation shown by Equation 4.19 can be rearranged to solve for the linear accelerations at the center of gravity.

$$\frac{\partial \mathbf{V}_{c.g.}}{\partial t} = \frac{\mathbf{F}}{m} - \vec{\Omega} \times \mathbf{V}_{c.g.} \tag{4.21}$$

The above equation can be broken into its scalar acceleration parts as shown in Equations 4.22 through 4.24.

$$\dot{u} = \frac{F_x}{m} - qw + rv \tag{4.22}$$

$$\dot{v} = \frac{F_y}{m} - ru + pw \tag{4.23}$$

$$\dot{w} = \frac{F_z}{m} - pv + qu \tag{4.24}$$

For a constant inertial system, the moment equation shown in 4.20 can be rearranged to solve for the angular accelerations.

$$\frac{\partial \vec{\Omega}}{\partial t} = \mathbf{I}^{-1}\left(\mathbf{M}_{c.g.} - \vec{\Omega} \times \left(\mathbf{I} \cdot \vec{\Omega}\right)\right) \tag{4.25}$$

where

$$\mathbf{I}^{-1} = \left|\mathbf{I}\right| \begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_4 & I_5 \\ I_3 & I_5 & I_6 \end{bmatrix} \tag{4.26}$$

with

$$\left|\mathbf{I}\right| = I_x I_y I_z - I_x I_{yz}^2 - I_z I_{xy}^2 - I_y I_{xz}^2 - 2 I_{yz} I_{xz} I_{xy} \tag{4.27}$$

$$\begin{bmatrix} I_1 & I_2 & I_3 \\ I_2 & I_4 & I_5 \\ I_3 & I_5 & I_6 \end{bmatrix} = \begin{bmatrix} I_y I_z - I_{yz}^2 & I_{xy} I_z + I_{yz} I_{xy} & I_{xy} I_{yz} + I_y I_{xz} \\ I_{xy} I_z + I_{yz} I_{xy} & I_x I_z + I_{xz}^2 & I_x I_{yz} + I_{xy} I_{xz} \\ I_{xy} I_{yz} + I_y I_{xz} & I_x I_{yz} + I_{xy} I_{xz} & I_x I_z + I_{xz}^2 \end{bmatrix} \tag{4.28}$$

As with the rearranged force equation, Equation 4.25 can be broken into scalar parts. The following three equations represent the first three of twelve state equations that are used to describe the motion of an aircraft in flight.

$$\begin{aligned} \dot{p} = \frac{1}{\left|\mathbf{I}\right|} & \Big(I_1 L + I_2 M + I_3 N + p^2\left(I_{xy} I_3 - I_{xz} I_2\right) + pq\left(I_{xz} I_1 - I_{yz} I_2 - \left(I_y - I_x\right) I_3\right) \\ & + pr\left(I_{yz} I_3 - I_{xy} I_1 - \left(I_x - I_z\right) I_2\right) + q^2\left(I_{yz} I_1 - I_{xy} I_3\right) + r^2\left(I_{xz} I_2 - I_{yz} I_1\right) \\ & + qr\left(I_{xy} I_2 - I_{xz} I_3 - \left(I_z - I_y\right) I_1\right)\Big) \end{aligned} \tag{4.29}$$

$$\begin{aligned} \dot{q} = \frac{1}{\left|\mathbf{I}\right|} & \Big(I_2 L + I_4 M + I_5 N + p^2\left(I_{xy} I_5 - I_{xz} I_4\right) + pq\left(I_{xz} I_2 - I_{yz} I_4 - \left(I_y - I_x\right) I_5\right) \\ & + pr\left(I_{yz} I_5 - I_{xy} I_2 - \left(I_x - I_z\right) I_4\right) + q^2\left(I_{yz} I_2 - I_{xy} I_5\right) + r^2\left(I_{xz} I_4 - I_{yz} I_2\right) \\ & + qr\left(I_{xy} I_4 - I_{xz} I_5 - \left(I_z - I_y\right) I_2\right)\Big) \end{aligned} \tag{4.30}$$

$$\dot{r} = \frac{1}{|\mathbf{I}|}\Big(I_3 L + I_5 M + I_6 N + p^2\big(I_{xy} I_6 - I_{xz} I_5\big) + pq\big(I_{xz} I_3 - I_{yz} I_5 - \big(I_y - I_x\big) I_6\big)$$
$$+ pr\big(I_{yz} I_6 - I_{xy} I_3 - \big(I_x - I_z\big) I_5\big) + q^2\big(I_{yz} I_3 - I_{xy} I_6\big) + r^2\big(I_{xz} I_5 - I_{yz} I_3\big) \tag{4.31}$$
$$+ qr\big(I_{xy} I_5 - I_{xz} I_6 - \big(I_z - I_y\big) I_3\big)\Big)$$

4.3 Flight Path Equations

In lieu of using the velocity variables *u*, *v*, and *w*, which are found in terms of the aircraft body axes, a set of axes based on the flight path reference system is used. The velocity used by the state equations then becomes the aircraft's true velocity, and the angle-of-attack α and the sideslip angle β are used to determine where the true velocity vector points with respect to the body axes. Figure 4.2 illustrates an aircraft and its flight path axes, it body axes, the corresponding angles, and the true velocity vector.

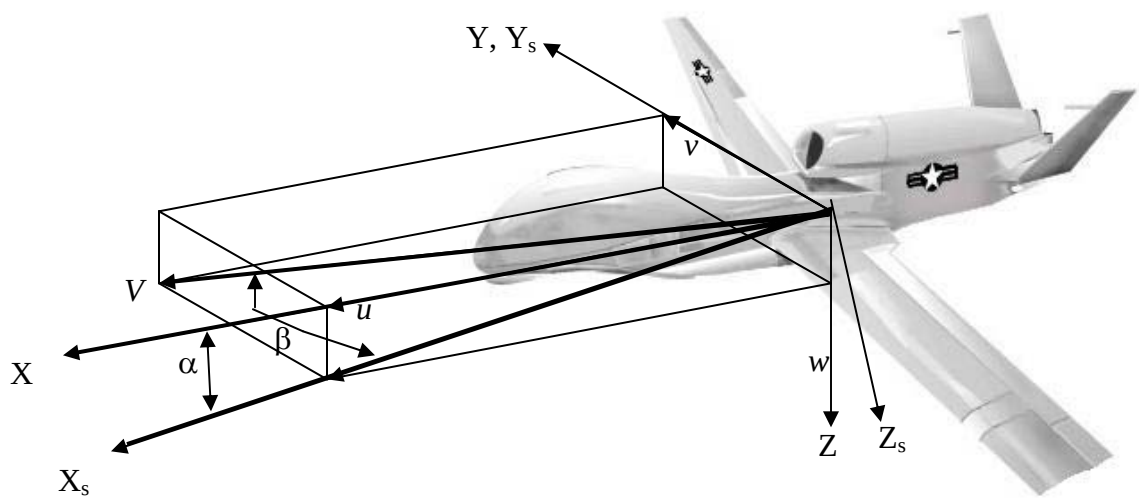

Figure 4.2: Stability axis system and angles with body axis system

Using this figure, it can be seen that the body axes-based velocities are related to the true aircraft velocity using:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = V \begin{bmatrix} \cos\alpha\cos\beta \\ \sin\beta \\ \sin\alpha\sin\beta \end{bmatrix} \tag{4.32}$$

The magnitude of the true velocity is then determined by the following equation.

$$V = \sqrt{u^2 + v^2 + w^2} \tag{4.33}$$

The angle-of-attack α and the sideslip angle β are then found by

$$\alpha = \arctan\left(\frac{w}{v}\right) \tag{4.34}$$

$$\beta = \arctan\left(\frac{v}{\sqrt{u^2 + w^2}}\right) \tag{4.35}$$

Determining the aircraft's true acceleration is accomplished by differentiating Equation 4.33, which results in:

$$\dot{V} = \frac{d}{dt}\left(\sqrt{u^2 + v^2 + w^2}\right) = \frac{u\dot{u} + v\dot{v} + w\dot{w}}{V} \tag{4.36}$$

Using the expressions for *u*, *v*, and *w* from Equation 4.32 yields:

$$\dot{V} = \frac{(V\cos\alpha\cos\beta)\dot{u} + (V\sin\beta)\dot{v} + (V\sin\alpha\sin\beta)\dot{w}}{V} \tag{4.37}$$

Finally, the fourth state equation can be found by substituting the expressions for the body axes accelerations found in Equations 4.22 through 4.24.

$$\dot{V} = \frac{1}{m}\left(F_x\cos\alpha\cos\beta + F_y\sin\beta + F_z\sin\alpha\sin\beta\right) \tag{4.38}$$

The fifth state equation is the rate of change of the angle-of-attack. It is determined by first differentiating Equation 4.34, as shown below.

$$\dot{\alpha} = \frac{d}{dt}\left(\arctan\left(\frac{w}{v}\right)\right) = \frac{u\dot{w} - \dot{u}w}{u^2 + w^2} \tag{4.39}$$

The above equation can be manipulated to get:

$$\dot{\alpha} = \frac{u\dot{w} - \dot{u}w}{V^2 - v^2} = \frac{u\dot{w} - \dot{u}w}{V^2 - (\sin\beta)^2} = \frac{u\dot{w} - \dot{u}w}{V^2 \cos^2\beta} \tag{4.40}$$

As with the true acceleration, the final form of the rate of angular change equation for angle-of-attack can be found by substitution of Equations 4.32 and 4.22 through 4.24.

$$\dot{\alpha} = \frac{1}{V\cos\beta}\left\{\frac{1}{m}\left(-F_x \sin\alpha + F_z \cos\alpha\right)\right\} + q - \left(p\cos\alpha + r\sin\alpha\right)\tan\beta \tag{4.41}$$

The sixth state equation is found in the same manner. The rate of change of the sideslip angle is first found by differentiating.

$$\dot{\beta} = \frac{d}{dt}\left(\arctan\left(\frac{v}{\sqrt{u^2 + w^2}}\right)\right) = \frac{\dot{v}\left(u^2 + v^2\right) - v\left(u\dot{u} + w\dot{w}\right)}{V^2\sqrt{u^2 + w^2}} \tag{4.42}$$

Substituting in the expressions for *u*, *v*, and *w* and their derivatives:

$$\dot{\beta} = \frac{1}{V}\left\{\frac{1}{m}\left(-F_x \cos\alpha\sin\beta + F_y \cos\beta - F_z \sin\alpha\sin\beta\right)\right\} + p\sin\alpha - r\cos\alpha \tag{4.43}$$

The forces and moments acting upon these first six state equations can be broken into components. These components consist of aerodynamic forces and moments, propulsion forces and moments, and gravitational force.

$$\begin{Bmatrix} F_x \\ F_y \\ F_z \end{Bmatrix} = \begin{bmatrix} X_{aerodynamic} & + & X_{propulsion} & + & X_{gravity} \\ Y_{aerodynamic} & + & Y_{propulsion} & + & Y_{gravity} \\ Z_{aerodynamic} & + & Z_{propulsion} & + & Z_{gravity} \end{bmatrix} \tag{4.44}$$

$$\begin{Bmatrix} L \\ M \\ N \end{Bmatrix} = \begin{bmatrix} L_{aerodynamic} & + & L_{propulsion} \\ M_{aerodynamic} & + & M_{propulsion} \\ N_{aerodynamic} & + & N_{propulsion} \end{bmatrix} \tag{4.45}$$

Typically, aerodynamic forces are used in more familiar terms of lift, drag, and side force as opposed to the body axis system forces. The two sets of forces are related by:

$$\begin{Bmatrix} X_{aerodynamic} \\ Y_{aerodynamic} \\ -Z_{aerodynamic} \end{Bmatrix} = \begin{bmatrix} -\cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ \sin\alpha & 0 & \cos\alpha \end{bmatrix} \begin{bmatrix} Lift \\ Drag \\ Y \end{bmatrix} \tag{4.46}$$

Aerodynamic forces and moments can be found using the following six equations.

In the longitudinal direction,

$$Lift = C_L \bar{q} S \tag{4.47}$$

$$Drag = C_D \bar{q} S \tag{4.48}$$

$$M = C_m \bar{q} S \bar{c} \tag{4.49}$$

In the lateral direction,

$$Y = C_Y \bar{q} S \tag{4.50}$$

$$L = C_l \bar{q} S b \tag{4.51}$$

$$N = C_n \bar{q} S b \tag{4.52}$$

Aerodynamic coefficients used in the above equations can be found from known aircraft coefficient derivatives. In the longitudinal direction, the coefficients are built up component-wise using the following three equations:

$$C_L = C_{L_0} + C_{L_\alpha}\alpha + C_{L_{iH}} i_H + C_{L_q}\left(\frac{q\bar{c}}{2V}\right) + C_{L_{\dot{\alpha}}}\left(\frac{\dot{\alpha}\bar{c}}{2V}\right) + C_{L_{\delta_E}}\delta_E \tag{4.53}$$

$$C_D = C_{D_0} + C_{D_\alpha}\alpha + C_{D_{iH}} i_H + C_{D_{\delta_E}}\delta_E \tag{4.54}$$

$$C_m = C_{m_0} + C_{m_\alpha}\alpha + C_{m_{iH}} i_H + C_{m_q}\left(\frac{q\bar{c}}{2V}\right) + C_{m_{\dot{\alpha}}}\left(\frac{\dot{\alpha}\bar{c}}{2V}\right) + C_{m_{\delta_E}}\delta_E \tag{4.55}$$

In the lateral direction, the coefficients are:

$$C_Y = C_{Y_\beta}\beta + C_{Y_p}\left(\frac{pb}{2V}\right) + C_{Y_r}\left(\frac{rb}{2V}\right) + C_{Y_{\delta_A}}\delta_A + C_{Y_{\delta_r}}\delta_r \tag{4.56}$$

$$C_l = C_{l_\beta}\beta + C_{l_p}\left(\frac{pb}{2V}\right) + C_{l_r}\left(\frac{rb}{2V}\right) + C_{l_{\delta_A}}\delta_A + C_{l_{\delta_r}}\delta_r \tag{4.57}$$

$$C_n = C_{Y_\beta}\beta + C_{n_p}\left(\frac{pb}{2V}\right) + C_{n_r}\left(\frac{rb}{2V}\right) + C_{n_{\delta_A}}\delta_A + C_{n_{\delta_r}}\delta_r \tag{4.58}$$

4.4 Earth-fixed Axes and Kinematic Relationships

The last six state equations are derived from a new set of axes and kinematic relationships. These equations will relate the aircraft orientation to an Earth-fixed set of axes. Figure 4.3 illustrates the principles discussed here for relating the aircraft to the Earth-fixed axes. The first step is to translate a set of axes parallel to those of the Earth-fixed axes until the origin of the translated set corresponds to the center of gravity of the aircraft. This set of axes will be labeled X_1, Y_1, and Z_1. These axes will be rotated three times to align themselves with the body axes of the aircraft.

The first rotation of the axes is about the Z_1 axis over the Euler angle ψ. This axis is then labeled X_2, Y_2, and Z_2. The next rotation of the new axes set is about the Y_2 axis through the Euler angle ϕ. This results in the new set of axes X_3, Y_3, and Z_3. The final

rotation of the axes is about the axis X_3, through the Euler angle θ. The set of axes that results from these three rotations is labeled X, Y, and Z, and is aligned with the body axes of the aircraft.

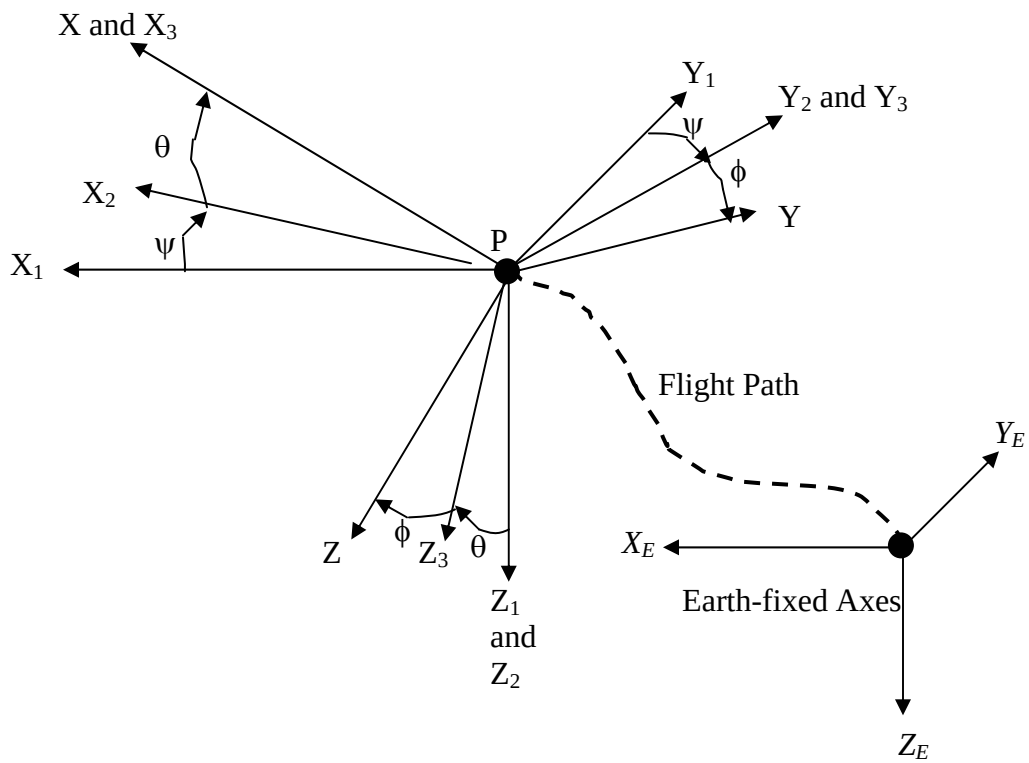

Figure 4.3: Aircraft orientation with Euler angles

The first relation from the above is that the first set of axes X_1, Y_1, and Z_1 is parallel to the Earth-fixed axis. From this, it is easily seen that

$$U_1 = \dot{x}_e \qquad V_2 = \dot{y}_e \qquad W_1 = \dot{z}_e \tag{4.59}$$

Using the above equation and relating each set of axis to the next, the Earth-relative velocities can be related to the body-relative velocities.

$$\begin{Bmatrix} \dot{x}_e \\ \dot{y}_e \\ \dot{z}_e \end{Bmatrix} = \begin{bmatrix} \cos\psi & -\sin\psi & 0 \\ \sin\psi & \cos\psi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \tag{4.60}$$

This reduces to provide an equation for each of the Earth-relative velocities.

$$\dot{x}_e = \{u\cos\theta + (v\sin\phi + w\cos\phi)\sin\theta\}\cos\psi - (v\cos\phi - w\sin\phi)\sin\psi \tag{4.61}$$

$$\dot{y}_e = \{u\cos\theta + (v\sin\phi + w\cos\phi)\sin\theta\}\sin\psi - (v\cos\phi - w\sin\phi)\cos\psi \tag{4.62}$$

$$\dot{z}_e = -u\sin\theta + (v\sin\phi + w\cos\phi)\cos\theta \tag{4.63}$$

The Z-axis is defined to be pointed downward, so the relationship between the Z_E axis and the altitude of the aircraft is:

$$\dot{h} = -\dot{z}_e \tag{4.64}$$

Using the expressions of Equation 4.32 to relate the body axes velocities to the true velocity V, the angle-of-attack α, and the sideslip angle β, the seventh, eighth, and ninth state equations are found to be:

$$\begin{aligned}\dot{x}_e = V\{&\cos\alpha\cos\beta\cos\theta\cos\psi + \sin\beta(\sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi) \\ &+ \sin\alpha\cos\beta(\cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi)\}\end{aligned} \tag{4.65}$$

$$\begin{aligned}\dot{y}_e = V\{&\cos\alpha\cos\beta\cos\theta\sin\psi + \sin\beta(\cos\phi\cos\psi + \sin\phi\sin\theta\sin\psi) \\ &+ \sin\alpha\cos\beta(\cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi)\}\end{aligned} \tag{4.66}$$

$$\dot{h} = V\{\cos\alpha\cos\beta\sin\theta - \sin\beta\sin\phi\cos\theta - \sin\alpha\cos\beta\cos\phi\cos\theta\} \tag{4.67}$$

The last three state equations come from the airplane kinematic equations. The relationship between the Euler angular rates and the angular velocity components is:

$$\vec{\Omega} = \mathbf{i}p + \mathbf{j}q + \mathbf{k}r = \dot{\vec{\psi}} + \dot{\vec{\theta}} + \dot{\vec{\phi}} \tag{4.68}$$

The Euler angular rates can be found by referencing which axis each rotates about. For the angular rate $\dot{\vec{\psi}}$, the rotation is about the Z_1 axis. This leads to the next equation.

$$\vec{\dot{\psi}} = \left(-\mathbf{i}\sin\theta + \cos\theta\left(\mathbf{j}\sin\phi + \mathbf{k}\cos\phi\right)\right)\dot{\psi} \tag{4.69}$$

The next angular rate is $\vec{\dot{\theta}}$, which rotates around the Y_2 axis.

$$\vec{\dot{\theta}} = \left(\mathbf{j}\cos\phi - \mathbf{k}\sin\phi\right)\dot{\theta} \tag{4.70}$$

The last angular rate is $\vec{\dot{\phi}}$. Since its rotation is about the X_3 axis, its equation is:

$$\vec{\dot{\phi}} = \mathbf{i}\dot{\phi} \tag{4.71}$$

These three relations can be substituted into Equation 4.68 to yield the kinematic equations:

$$\begin{Bmatrix} p \\ q \\ r \end{Bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \cos\theta\sin\phi \\ 0 & -\sin\phi & \cos\theta\cos\phi \end{bmatrix} \begin{Bmatrix} \dot{\psi} \\ \dot{\theta} \\ \dot{\phi} \end{Bmatrix} \tag{4.72}$$

The final three state equations are found by inverting the above equation.

Invert to get:

$$\dot{\psi} = \sec\theta\left(q\sin\phi + r\cos\phi\right) \tag{4.73}$$

$$\dot{\theta} = q\cos\phi - r\sin\phi \tag{4.74}$$

$$\dot{\phi} = p + \left(q\sin\phi + r\cos\phi\right)\tan\theta \tag{4.75}$$

At this time, all twelve state equations have been developed, and an aircraft model can be implemented into the simulation presented in the next chapter.

Chapter 5
Development of Centralized UAV Simulation

5.1 Main Simulation System

The simulation environment developed in this chapter is one where a central processor controls all of the decision-making abilities for the entire UAV team. This simulation fulfils the third and fourth research objectives as presented in Chapter 1, and is time-varying since the states of targets can change, UAVs can be and actually are destroyed during the simulation, unknown threats and targets can appear, and the group of UAVs can replan using this new information.

Figure 5.1 shows the main SIMULINK block of the simulation code.

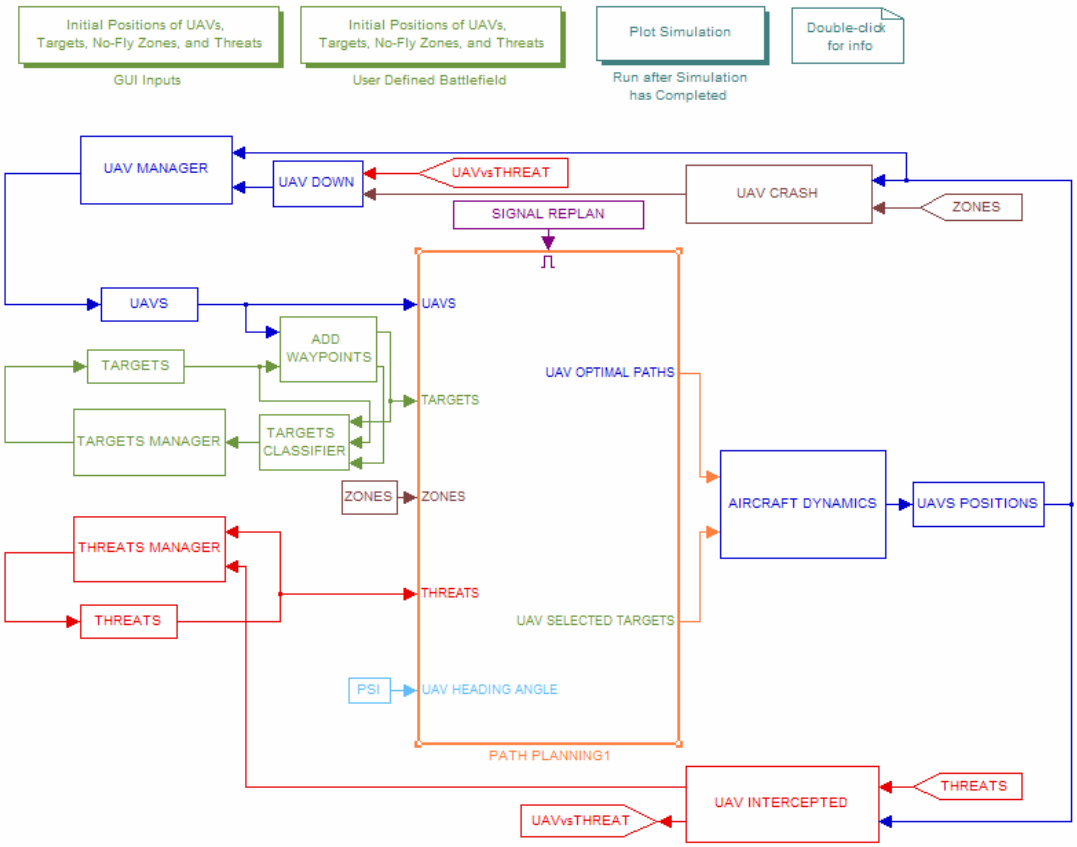

Figure 5.1: Main simulation system

There are several main components of the simulation, and each will be discussed separately in subsequent sections of this chapter. The first component is the simulation inputs. All necessary information is inputted based on graphical user interfaces that are discussed in Section 5.2. The top left block initializes these. The next component is the path planning and task allocation block, seen as the large middle block of Figure 22. This block executes the path planning and task allocation approach discussed in Chapter 3 and will be elaborated further upon is Section 5.3. Once a UAV is given an objective and has a planned path, the aircraft dynamics discussed in Chapter 4 are implemented in the Aircraft Dynamics Blockset, discussed in Section 5.4. The outputs from this are positions of each UAV, which are checked to see if the position coincides with a threat boundary or a no-fly zone. If a UAV position does meet one of these criteria, another scheme is executed to determine if the UAV is destroyed or survives. A UAV Manager block is discussed in Section 5.5. This block keeps track of all UAVs and triggers action to be taken if a UAV is lost. The Targets Manager block keeps track of the state of each target. As assignments are completed by individual UAVs, target states change, and targets are eventually removed once confirmed as destroyed. More information about the Target Manager is found in Section 5.6. The last main block of the simulation is the Threats Manager. It is similar to the Targets Manager, and keeps track of all known threats, their positions, and when they fire at a UAV. Section 5.7 will complete the discussion of this manager. The final section of this chapter shows the outputs of the simulation and gives an example simulation.

5.2 Simulation Inputs

The obvious first step for the simulation is to initialize all inputs. The necessary inputs can be derived from the original problem statement given. The first information is the number of UAVs, targets, threats, and no-fly zones. Because the fourth research objective states that the simulation should be of a dynamic environment, the targets and threats are divided into the number of static and the number of pop-up for each. Pop-up targets and threats are now defined as those that are not known by the UAV team

whenever the simulation first begins, but rather appear after a time that the omnipotent user defines.

Graphical user interfaces have been developed to collect this necessary information in an easy manner. Figure 5.2 illustrates the main menu of the GUIs.

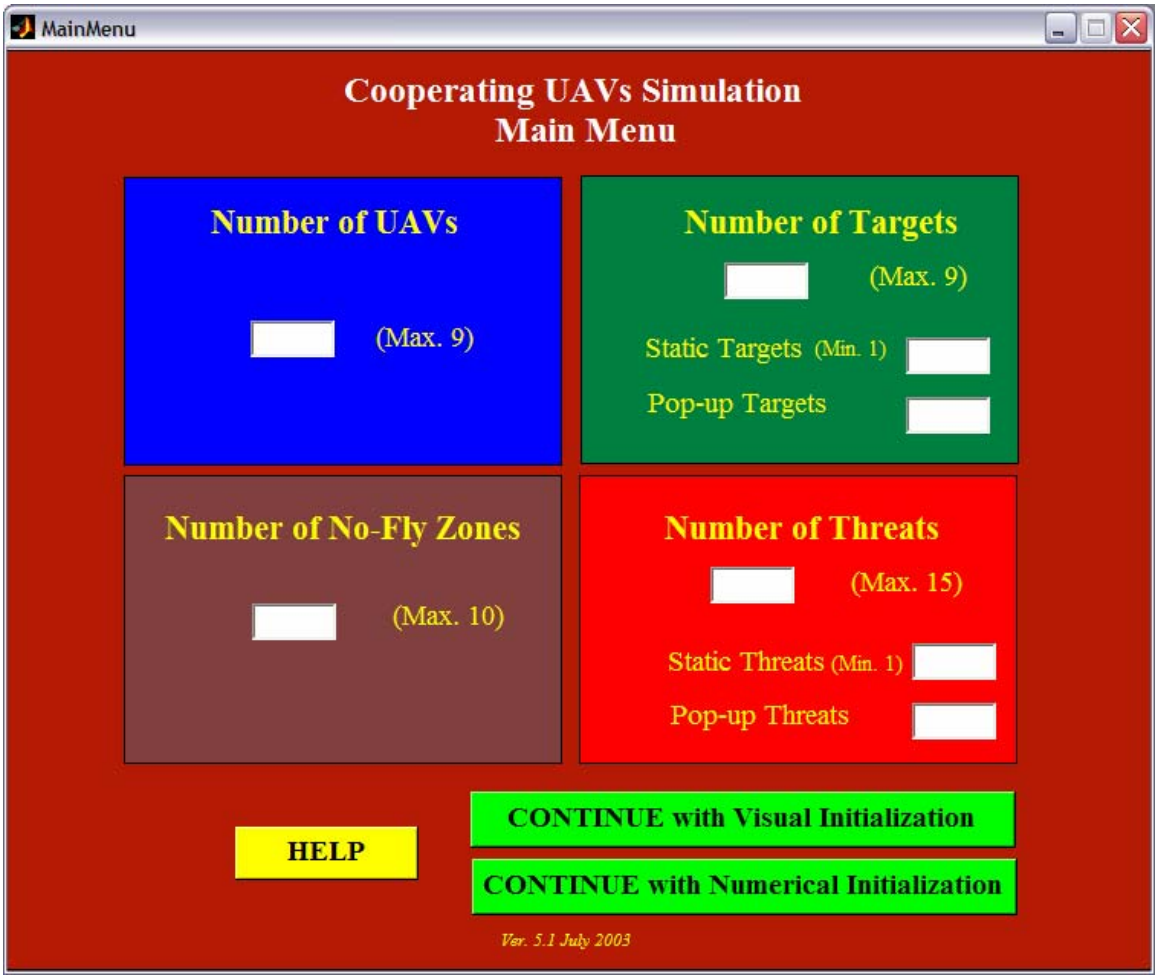

Figure 5.2: Cooperating UAVs Simulation Main Menu

This GUI collects the information specified above and allows the user to continue inputting information in one of two ways. The first way, the GUIs with visual initialization, will be discussed further in this section and allow the user to 'point-and-click' to initialize the battlefield. The second way to input the information is with the numerical initialization GUIs, where the user types in all locations manually.

All GUIs in this initialization scheme have error checking. All numbers inputted must be within proper ranges, and all necessary values must be specified for proper initialization. The following figure shows the error message shown to the user whenever an input error is detected.

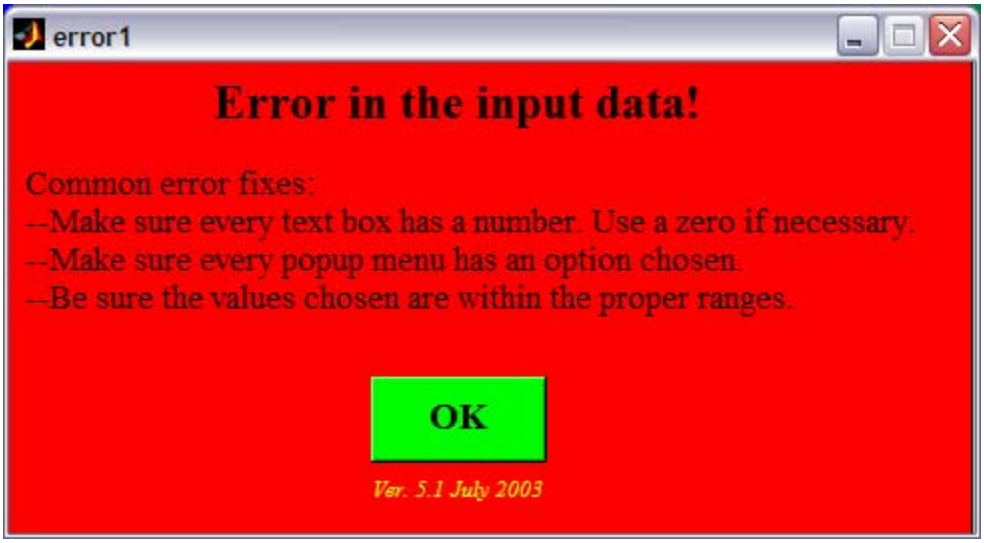

Figure 5.3: Error message

The next step for initializing the data is the aircraft menu. From the previous chapter, it is clear that many aircraft parameters are needed to accurately model the aircraft dynamics. The following menu uses several 'built-in' aircraft with all the necessary parameters already defined. The only necessary input from the user is the type of aircraft and its initial positions. Using the numerical initialization option, a value for each Earth-fixed initial position is chosen manually by entering numbers. With the graphical initialization option, only the height needs to be typed for the aircraft position. Figure 5.4 illustrates the Aircraft Menu GUI.

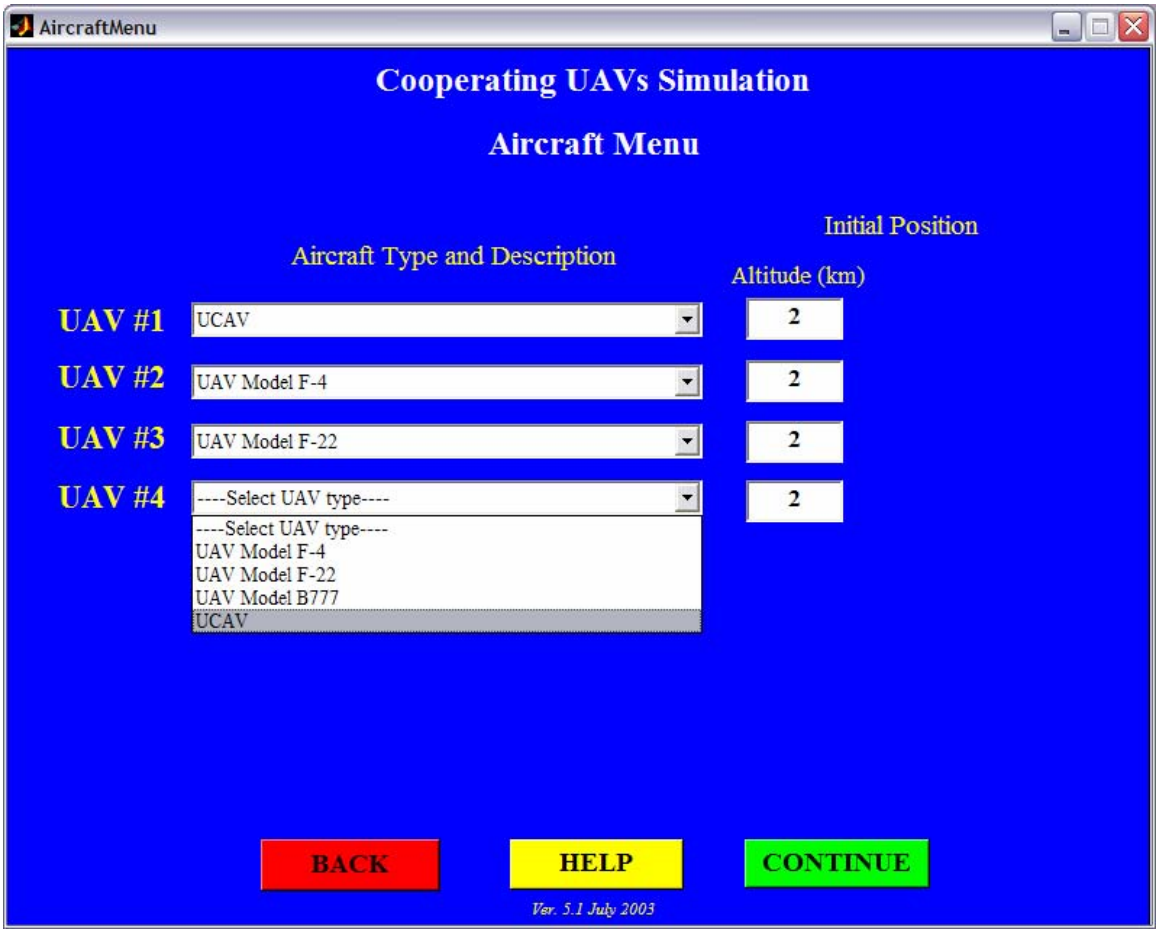

Figure 5.4: Aircraft Menu GUI

The Earth-fixed axial positions for X_E and Y_E are now entered using a graphical 'point-and-click' method. A message indicating what is being placed on the graph is displayed, along with instructions to first determine the location of the object using the crosshairs, and finally click on that location to place the object there. Figure 5.5 shows the use of this point-and-click tool for setting up the battlefield. This method is quite useful for determining where appropriate locations for the UAVs should be and illustrates where the placed UAVs for use in placing other UAVs.

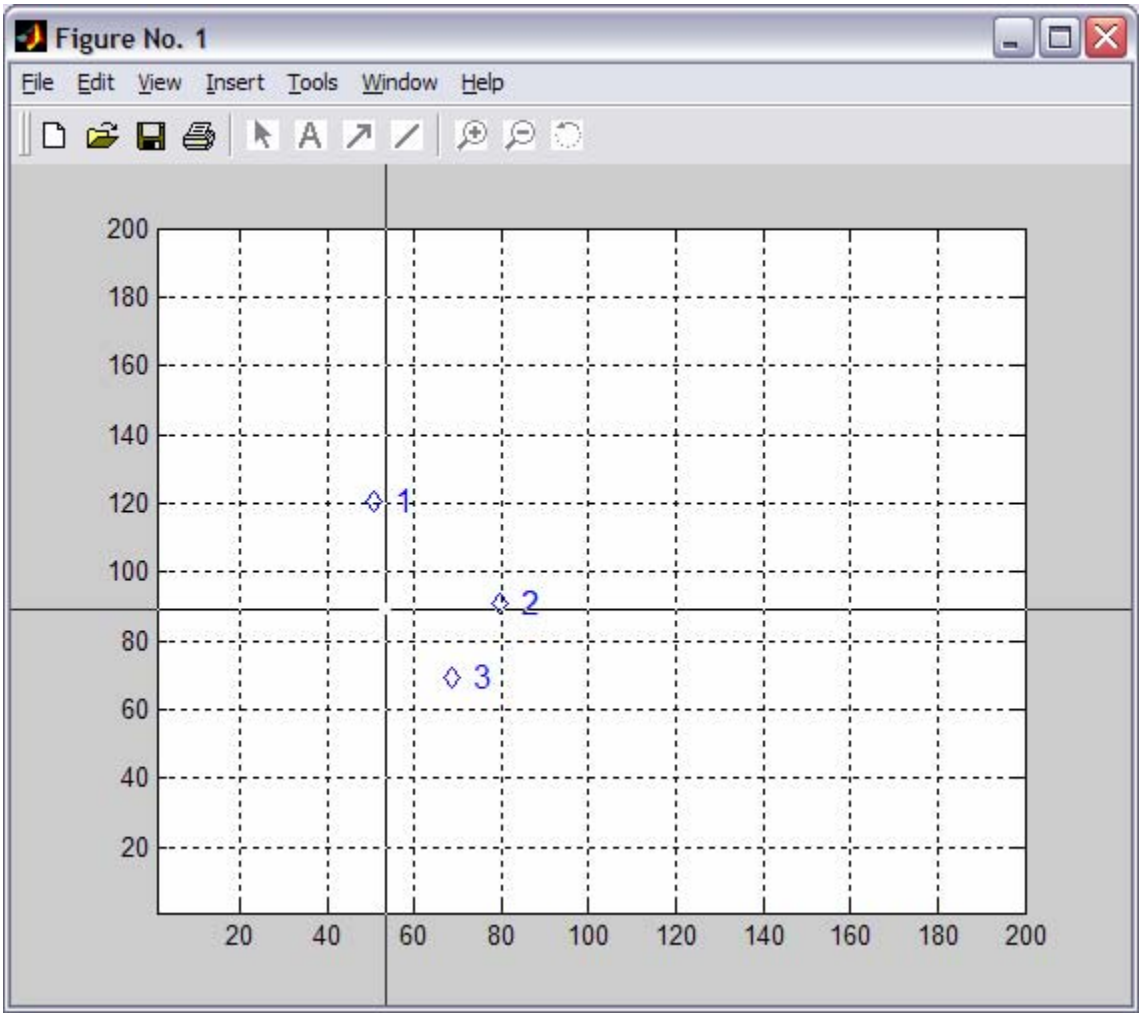

Figure 5.5: Point-and-click method of placing UAV positions

The UAVs are displayed as blue diamonds with the individual number to the right of the UAV. The battlefield size is defaulted to a 200-kilometers by 200-kilometers. This size was selected so that longer distances for target engagement could be simulated without having an excessively long simulation time for literal cross-country travel by a team of UAVs.

Next, the target information is inputted. Two different menus are used to accomplish this task. The first of these menus is for the static target values and locations, while the second is for the so-called 'pop-up' target values and locations. Since the typical mission envisioned within this research has been the 'high-risk but high-value'

mission type, associated values for individual targets are appropriate. These values are used for determining which targets are attacked first in the case of more targets than UAVs, and will be discussed further in the next section. The Static Target Menu and the Pop-up Target Menu vary only by addition of a pop-up time for the second menu.

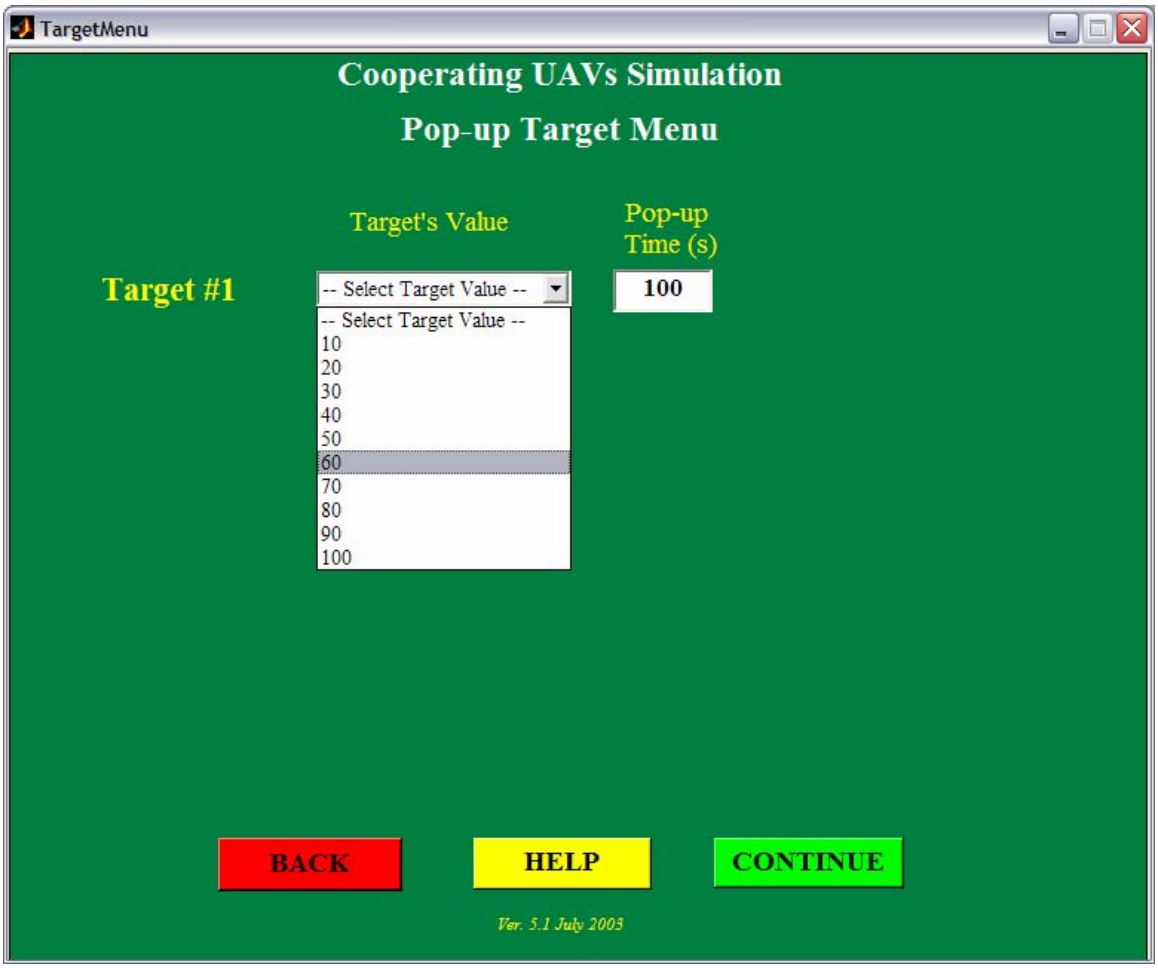

Figure 5.6: Pop-up Target Menu

The static target values are first selected; next, the static targets are then placed using the same 'point-and-click' method as discussed earlier. The UAV positions are still visible when targets are placed for ease of battlefield setup. Once the static targets are placed, the pop-up menu is used to select values for pop-up targets. These targets are then placed to complete the target information for the simulation. On the battlefield plot, static targets are depicted by a green 'x', while the popup targets are shown with a green cross.

Now, no-fly zone information is required. The only necessary information for these is the location and the radius. Figure 5.7 is the menu for the radius input. No-fly zones can represent two either physical or political boundaries that the UAVs are not allowed to cross. For ease of use input, the no-fly zones are modeled as simple mountains with a known radius. While input of complex political boundaries could be accomplished, it was chosen that 'point-and-click' mountains would be used to simulate no-fly zones.

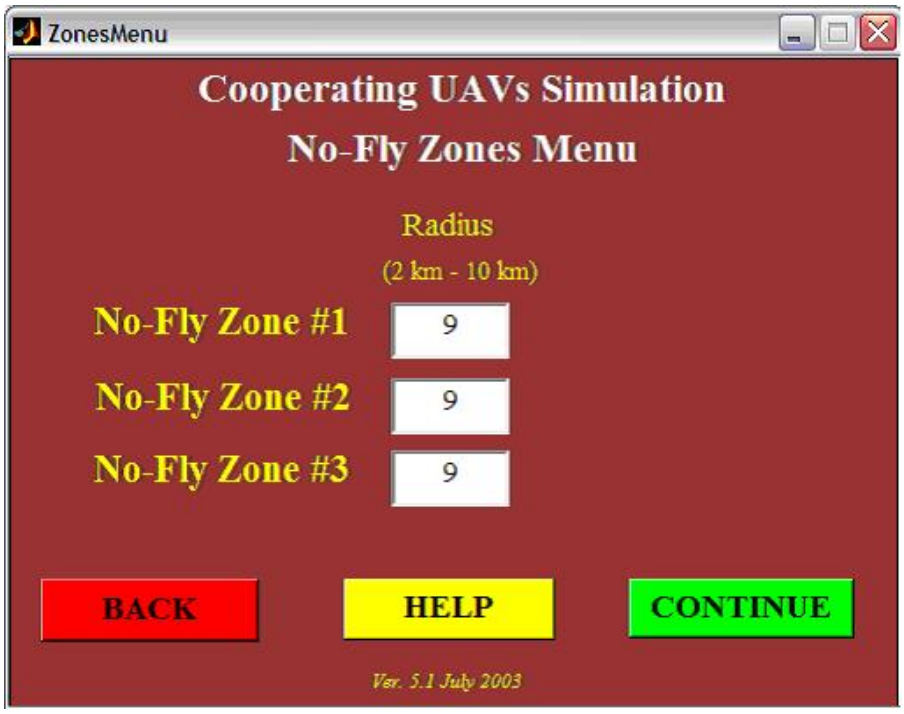

Figure 5.7: No-Fly Zones Menu

Once the radius of each no-fly zone is inputted, the point-and-click menu appears and the locations of the no-fly zones are chosen. On the battlefield plot, each placed no-fly zone appears as a black filled in circle of given radius. The previously placed UAVs and targets are also visible on the plot while no-fly zones are placed.

The last inputs are the descriptions and locations of the threats. Threats are broken into two groups of static and pop-up, similar to the targets. Typical threats are

built-in to the drop down list for the threat type and description. The description includes the effective range of the threat and the probability of kill. The threats that are built in to the list include all the threats described in Table 3.1 from Chapter 3. As with the target menus, the static threat information is first input and then locations are point-and-click inputted. The static threats appear on the battlefield as a red star with a red circle of effective range surrounding.

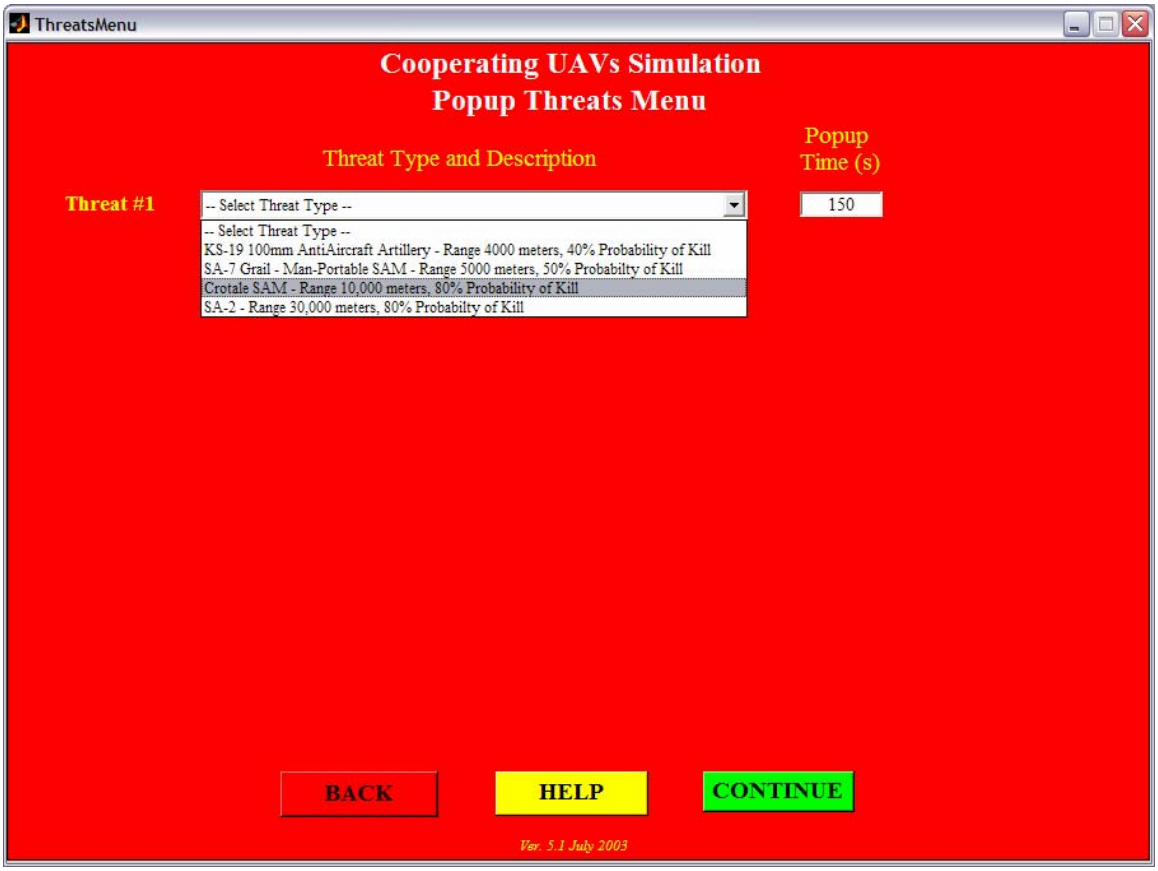

Figure 5.8: Pop-up Threats Menu

The 'Pop-up Threats' menu also includes the pop-up time for each threat. As above, the threats are then placed where desired on the battlefield. All previously placed objects will still be visible when placing the threats. Pop-up threats will appear as red 'O's with a red circle of the effective range surrounding it.

At this point, all needed information is now entered into the simulation. A typical final battlefield setup is shown in Figure 5.9, below. The next sections will describe the components used within the simulation.

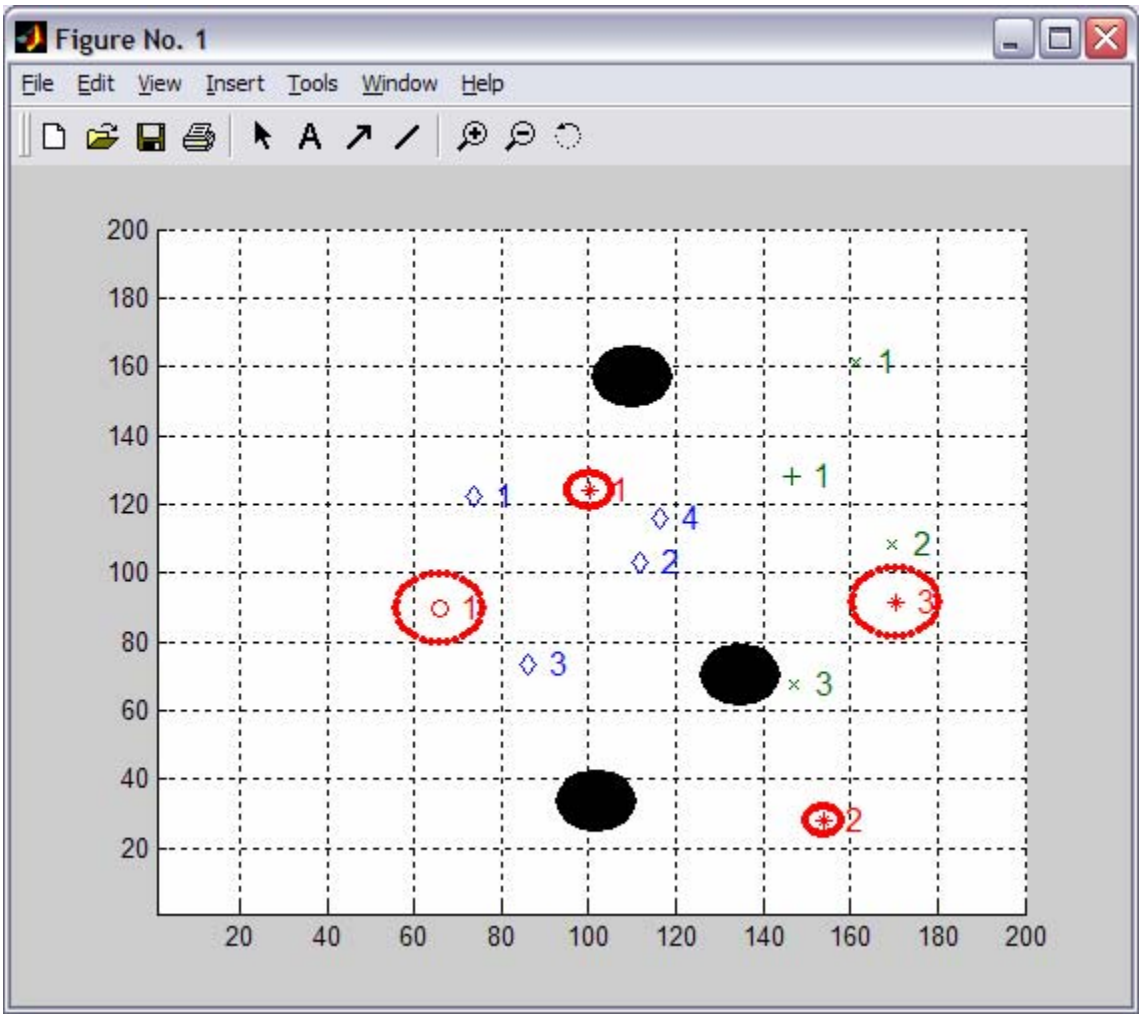

Figure 5.9: Example battlefield setup

5.3 Path Planning and Task Allocation Execution

Before the path planning and task allocation scheme can be executed, the number of targets and waypoints must first be equated with the number of UAVs. This requirement is a consequence of the MMKP constraints that each UAV must be assigned a single task and each target is required to be visited. However, this is fairly easily

overcome using the Place Waypoints block and the accompanying MATLAB code, *place_waypoints.m*, as found in Appendix B.

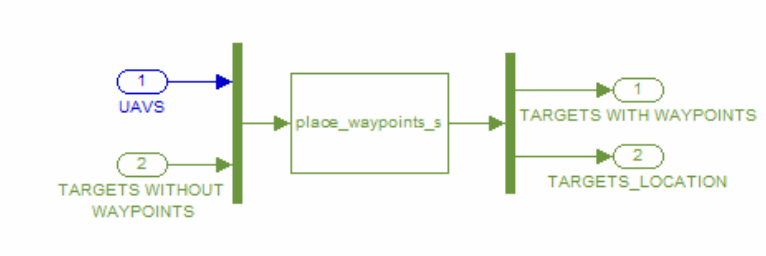

Figure 5.10: Place Waypoints block

The UAV locations and velocities and the target locations, values, and states are input into the block. The MATLAB code *place_waypoints.m* is then executed. This code approaches the problem with two different types of solution. For the situation where there are more targets than UAVs, the program sorts the targets by the highest values and removes the lower-valued targets for any number greater than the number of UAVs. The removed, lower-valued targets are not forgotten and will be later added back in to the list of targets whenever higher-valued targets are removed after being destroyed. The second solution is employed whenever the number of UAVs is higher than the number of targets, such as toward the end of a mission. In this situation, waypoints are added to the list of targets until the number of targets equals the number of UAVs. These waypoints are added at the same locations as the targets based upon the value of targets. Targets with a higher value have waypoints added to their position before lower-valued targets. This is to help ensure that higher-valued targets will have a higher probability of successful mission accomplishment by assigning multiple UAVs to these targets.

The actual method of assigning waypoints begins by finding the highest valued target. The location and value of this target are then stored and a waypoint is assigned with the same location but no value. The stored value is then decreased by 50%. The reason that the stored value decreases in half is that if the target is more than twice as valuable as any other target, it will automatically get two waypoints assigned to it before any other target gets an extra waypoint. The waypoints do not have values themselves

because they are simply the same as the target whose coordinates they share. The program executes for the same number of times as the difference between the number of UAVs and targets.

The path planning and task allocation scheme can now be executed. The following block diagram illustrates the inputs for the scheme and the outputs

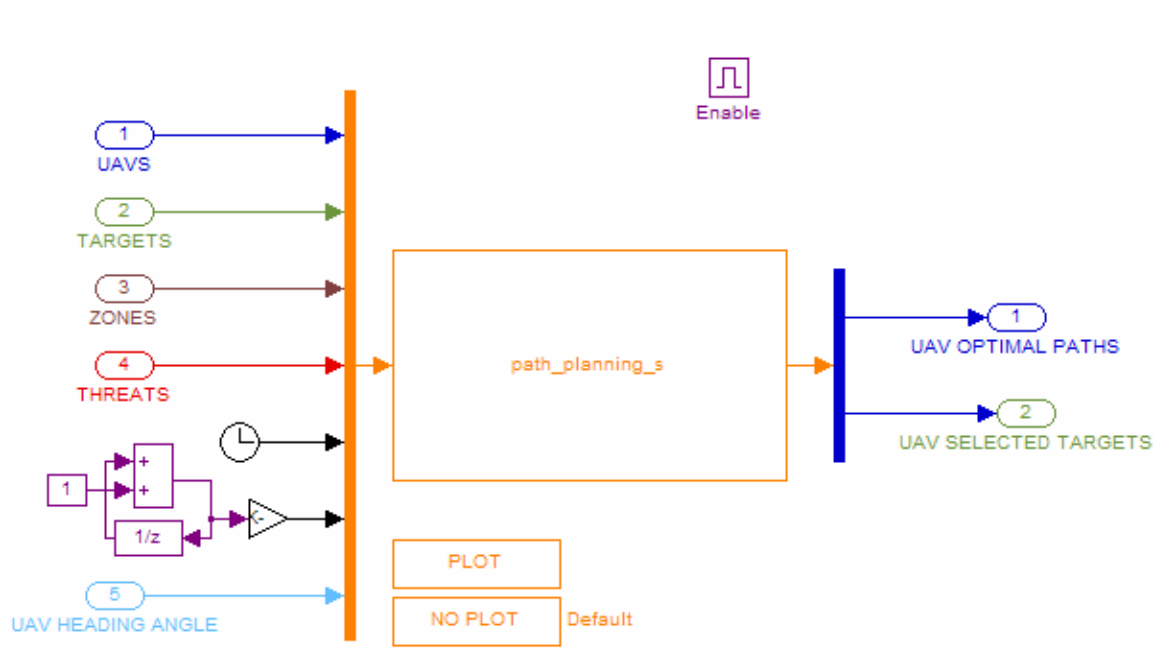

Figure 5.11: Path Planning and Task Allocation block

Inputs into this block are the following:

- UAV coordinates, altitudes, velocities, and heading angles
- Target coordinates
- No-fly zone coordinates and radii
- Target coordinates, effective ranges, and probability-of-kill
- The time at which the program is executing
- The number of times the path planning and task allocation scheme has executed

The program then executes and outputs which assignment each UAV receives and the corresponding optimal path for the UAV to fly to complete that assignment. Options are also given whether the user wants to see static plots for every execution of this block. It should be noted that this block contains an “Enable”, seen at the top of Figure 5.11. This addition indicates the path-planning scheme will only execute whenever the Enable is triggered. When the simulation is first started, the path planning and task allocation scheme will execute, but after that only when there is a signal to indicate replan. The necessary conditions to produce a replan are discussed in later sections of this chapter.

5.4 Aircraft Dynamics Subsystem

A six degree-of-freedom aircraft model is used within this section to model the aircraft dynamics. The centralized control scheme simulates all of the UAV dynamics for the entire group. The inputs to this section are specifically the outputs of the previous section, the assignment each UAV receives and the corresponding optimal path for the UAV to fly to complete that assignment. The outputs are the current positions and rotations of the UAV, the current heading angle of the aircraft, and an end-of-path signal for each UAV (to indicate when it has reached the target).

Figure 5.12 shows the ‘UAV Dynamics’ block for each of the possible UAVs, and the inputs and outputs of the block. Note that while there are blocks for 9 UAVs, there do not have to be 9 UAVs in the simulation, only a maximum of 9 UAVs. If there is less than the maximum number of UAVs running in the simulation, whether from the initialization or due to UAV loss, the individual blocks are not enabled within the centralized simulation. All of the present UAVs will then contribute to the outputs of positions and rotations, the heading angle output, and the end-of-path signals.

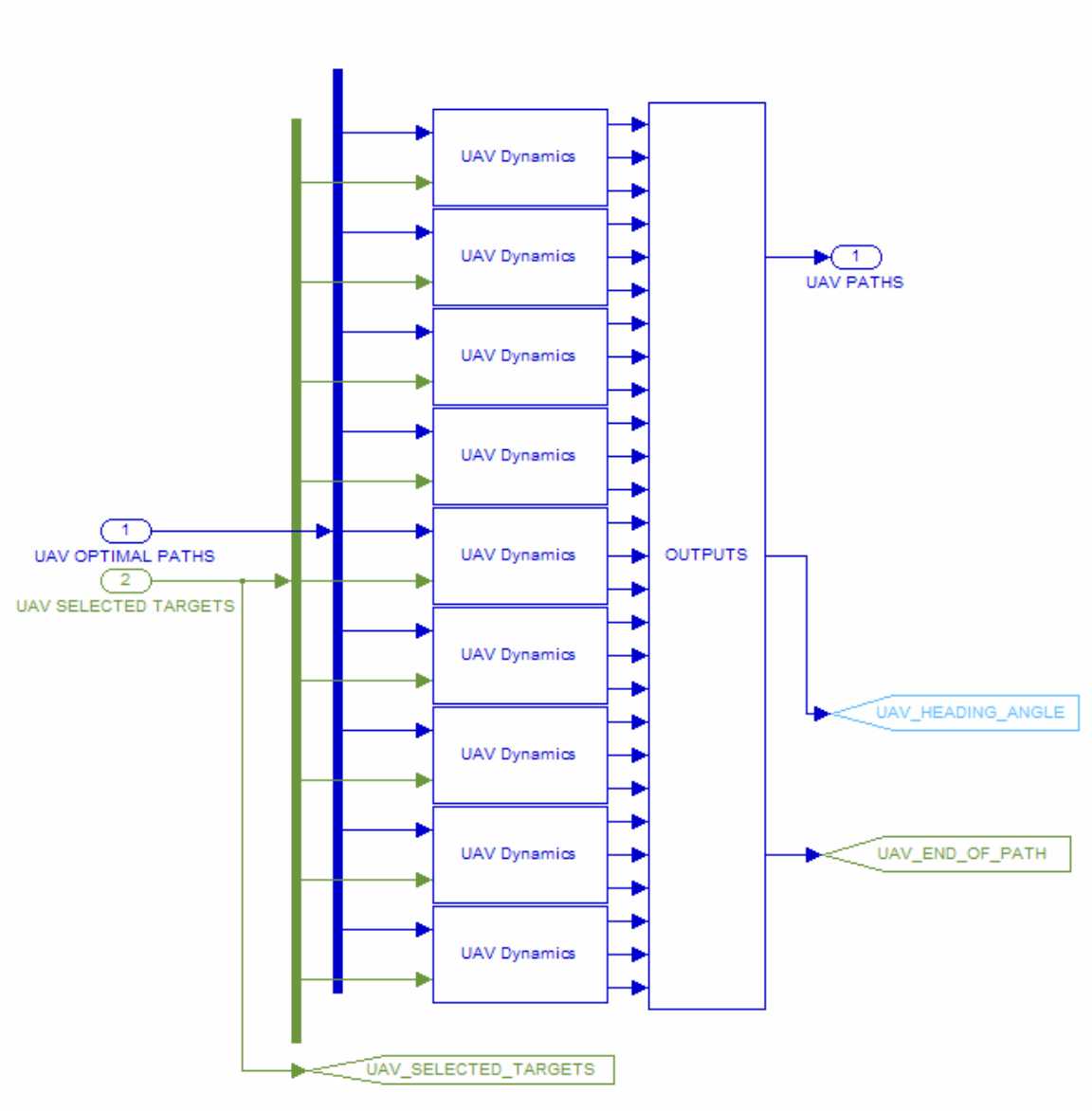

Figure 5.12: ‘UAV Dynamics’ blocks for all UAVs

Under each of the blocks labeled ‘UAV Dynamics’ lies the subsystem shown in Figure 5.13. This block coordinates the enabling of the aircraft model is the UAV is present, or if the UAV is not involved in the simulation, the appropriate outputs to indicate this.

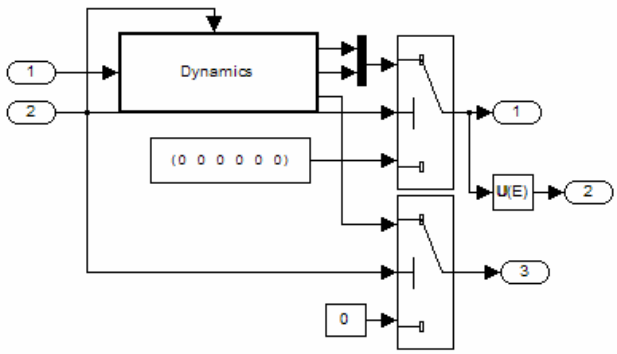

Figure 5.13: Blocks to output UAV positions, heading angle, and signal end of path

The above subsystem sends an enable signal to the blocks show in Figure 5.14. These blocks are subsystems for three separate functions. The first mask labeled 'X, Y, Z, time, pos_des' is used to determine a next position for the individual UAV. The 'End of path' block is used to determine when the UAV has reached the target position, and the 'UAV DYNAMICS' block is a mask for the actual aircraft model and autopilot subsystem.

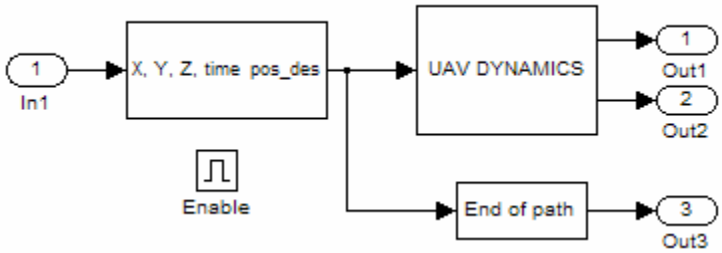

Figure 5.14: Determines next path position, runs aircraft model, and signals end of path

The first of these subsystems to be discussed is the 'X, Y, Z, time, pos_des' block. Looking under the mask results in the blocks shown in Figure 5.15. These blocks are used to break up the paths coming out of the path planning and task assignment scheme into short segments to use with the aircraft model. This is accomplished by using look-up

tables to find where the UAV will be on the path after a small elapsed amount of time. This location is then outputted and used with the aircraft model.

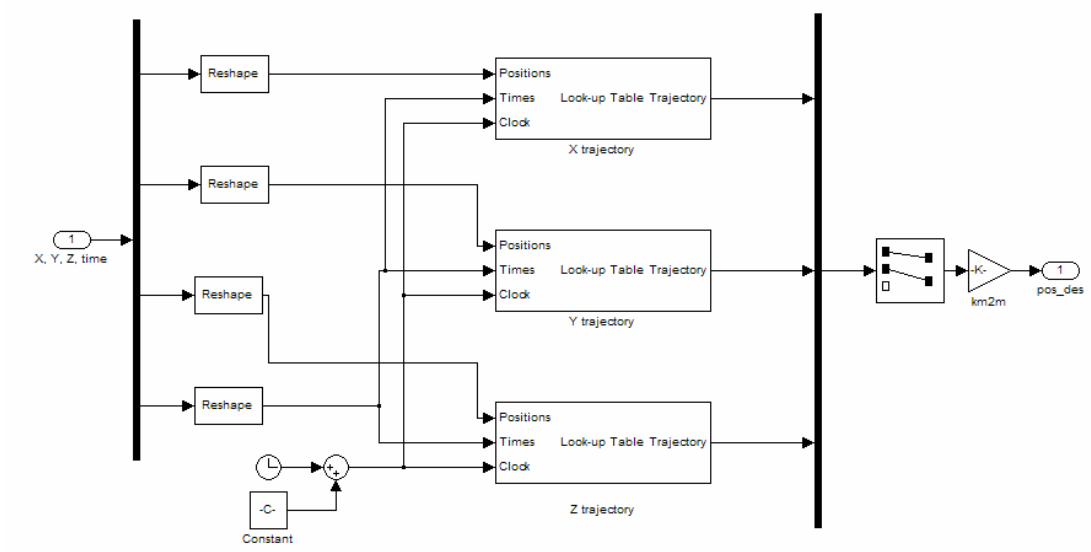

Figure 5.15: Blocks that 'look ahead' and output next position in path

The next mask cover the simple subsystem used for determining when a UAV has reached the end of its path, which is analogous to saying the UAV has reached its target. Whenever the UAV reaches its target, it no longer can look forward in time to the next position on its assigned path. This causes an empty output, which signals the target has been reached.

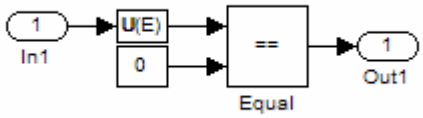

Figure 5.16: Determination of end of assigned path

The last of these three subsystems is the actual subsystem that controls the aircraft motion. Figure 5.17 illustrates this subsystem.

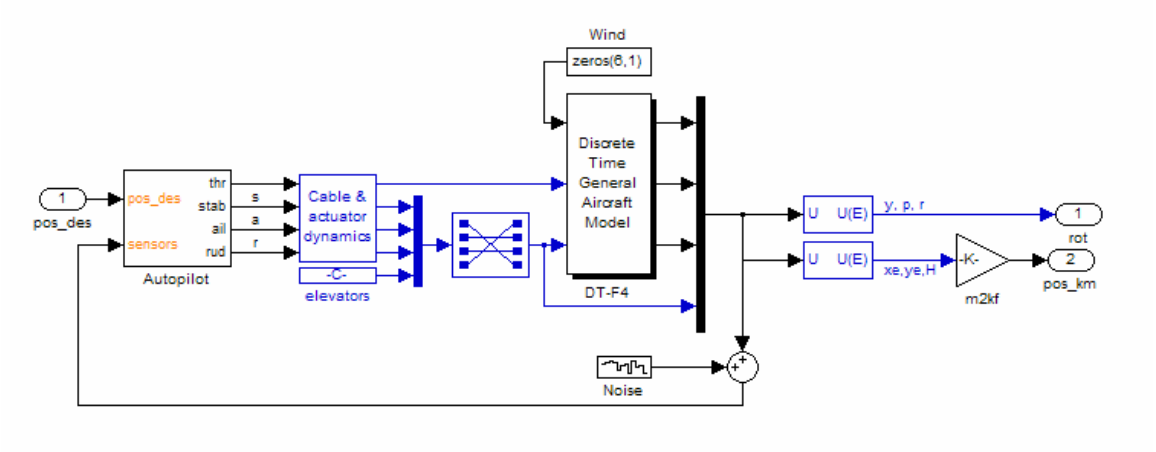

Figure 5.17: Actual UAV dynamics block, with aircraft model and heading-angle autopilot

This subsystem itself contains three major subsystems. The first and most obvious system is the block labeled "Discrete Time General Aircraft Model'. This is where specific control commands are inputted and used in conjunction with external forces and moments and known aircraft parameters to model the aircraft dynamics. This flight simulation environment is an open-source blockset distributed as FDC (*Flight Dynamics and Control*) [49]. This environment consists of five groups, which can be viewed in Figure 39. The first such group is the Airdata group. This group contains the standard atmospheric model, such as gravity variation, temperature, pressure, density, and equations related to these, such as dynamic pressure and Mach number. The second group is the Aerodynamics group. This group calculates the dimensionless coefficients discussed in the fourth chapter, in Equations 4.53 through 4.58. The third group calculates forces associated with gravity and wind, and are used in conjunction with Equations 4.44 and 4.45. The fourth group is the Aircraft Equations of Motion group. This group uses the twelve state equations in conjunction with the first three blocks to completely describe the motion of the aircraft. These state equations are solved using a fourth-order Runge-Kutta method. The last group is the Additional Outputs group. Contained within this group is the determination of the flight path variables, the time-derivatives of the body axes velocity components and acceleration components, and the grouping of aerodynamic forces and moments, propulsive forces and moments, gravity forces, and atmospheric turbulences.

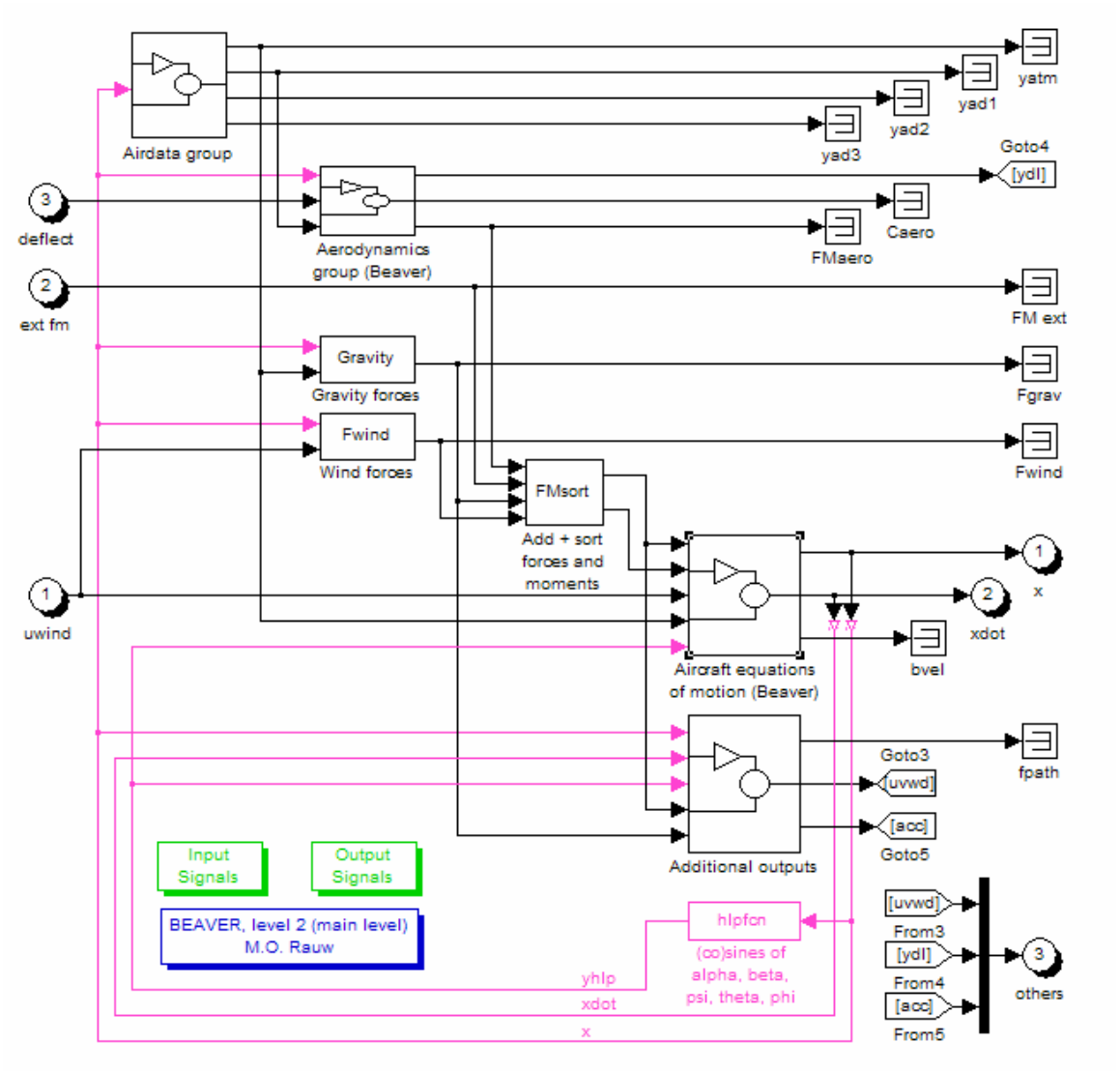

Figure 5.18: Flight simulation environment for aircraft model

The aircraft parameters seen in Figure 5.19 are used with the flight simulation environment to model the motion of the aircraft. These parameters include the geometry, mass, and inertial properties, aerodynamic coefficient derivatives, and the state vector of initial conditions that was shown in Equation 4.1. These parameters can be set up to be entered manually, as shown in the figure or can be used in conjunction with the specific aircraft selected from the GUI inputs.

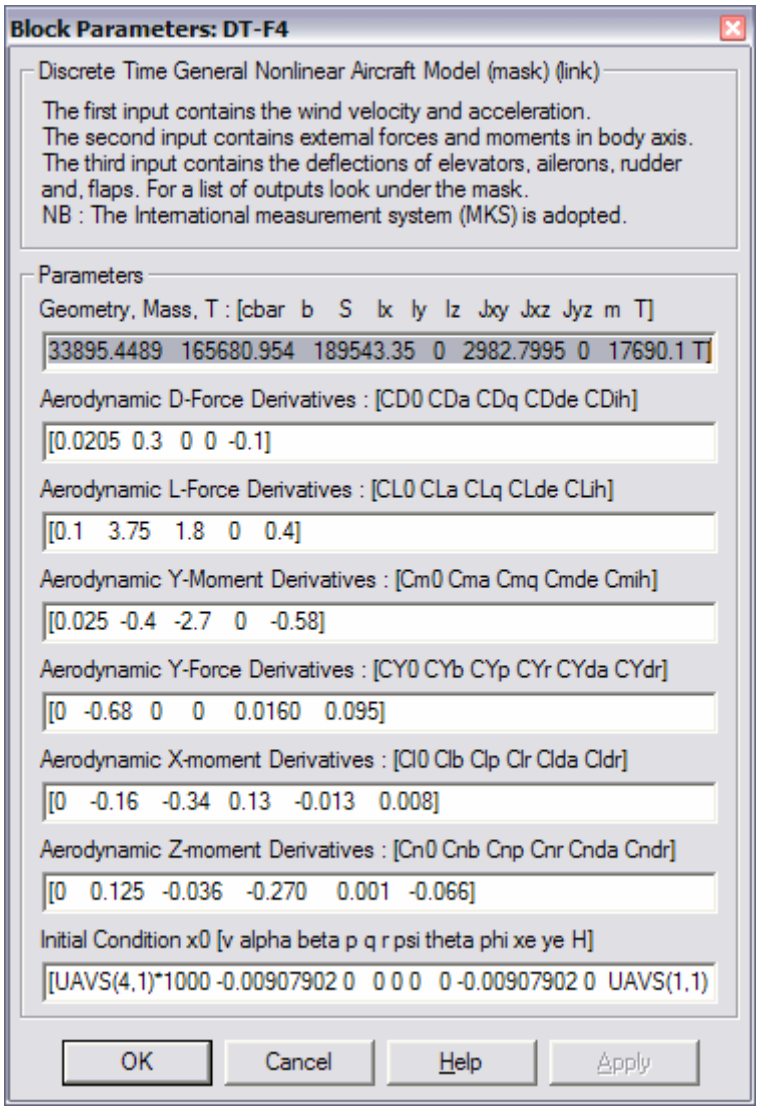

Figure 5.19: Parameters and inputs for aircraft model

The second subsystem shown in Figure 5.17 is the Cable and Actuator Dynamics subsystems. This system models the dynamic response associated with the throttle, stabilators, ailerons, and rudder as generic first order systems with an inherent delay.

$$G_{Actuator}(s) = \frac{a}{s+a} \tag{5.1}$$

The ailerons are modeled as a fast response system with the value of *a* set to 40. The rudder and stabilators are modeled as moderately fast actuators with the value of *a* set to 15. The throttle is set to a slow response, with a low value of 4 being used for *a*.

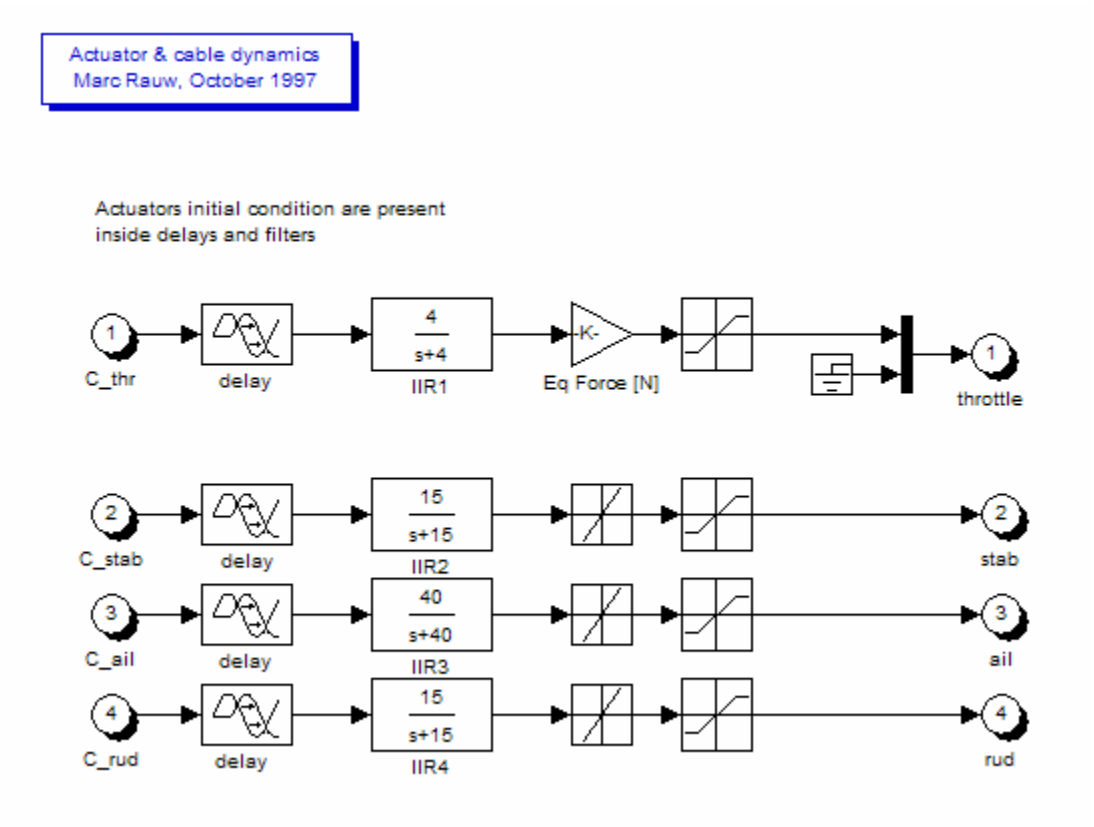

Figure 5.20: Actuator and cable dynamics subsystem

The third subsystem shown is the heading angle autopilot. This autopilot generates commands in terms of throttle adjustment and stabilators, aileron, and rudder deflections to follow a desired heading angle. This is where the input of looking ahead in the path is used. The aircraft compares its current position and rotations with those of where it needs to be at in certain amount of time (usually 15 or 20 seconds later). It then uses the autopilot shown in Figure 5.21 to generate the necessary commands to follow that path (or at least attempt to in case that the path is not dynamically feasible).

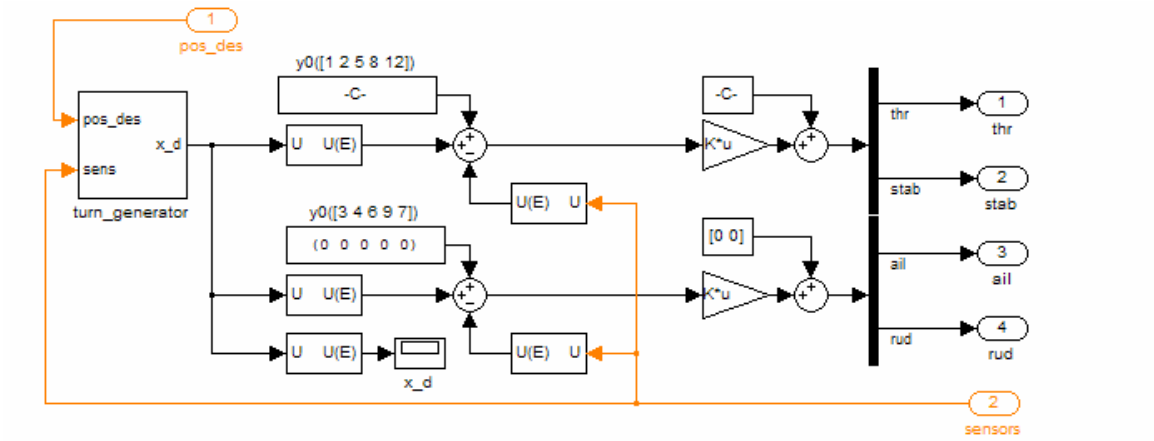

Figure 5.21: Heading angle autopilot, showing turn generator

The turn generator of Figure 5.21 is shown in detail in the below figure. This system generates the necessary outputs of p, q, r, and the Euler angles of ψ, θ, and ϕ.

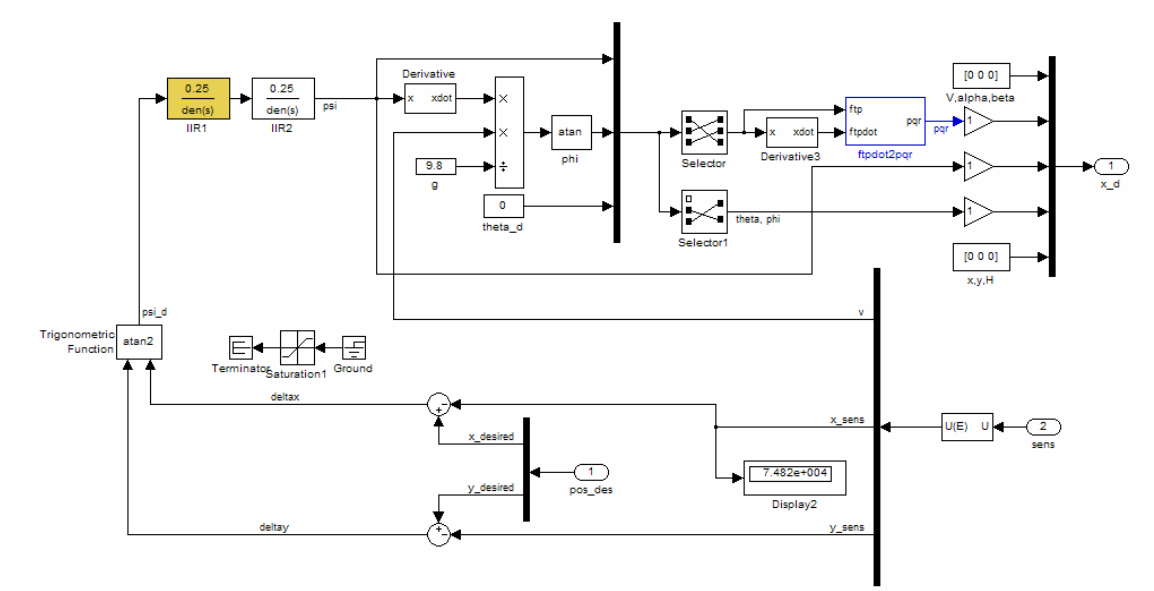

Figure 5.22: Turn generator subsystem

This completes the discussion of the modeling of the aircraft dynamics. The last part of this section is the block called UAV Positions in the main system. This block removes the angular orientations of the aircraft and leaves only the positions of each UAV for use in later calculations of the simulation. Heading angle is the only orientation

angle that is used for the path-planning scheme, and it is output before reaching this block. The other orientation angles are not needed for such calculations as if the UAV is destroyed or when a UAV reaches the end of its path. However, all state information is contained within the system of Figure 5.17 for each individual UAV, so these angular orientations are not lost, just removed from the UAVs matrix.

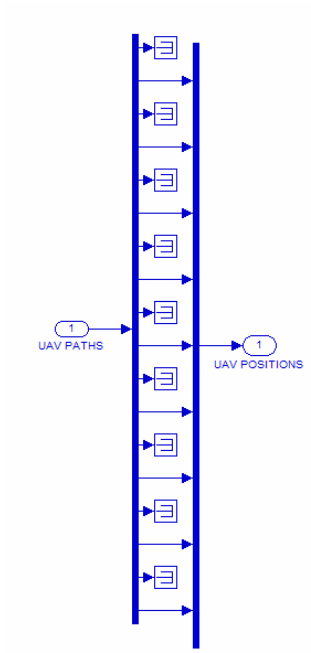

Figure 5.23: UAV Positions block

5.5 UAVs Manager

For the centralized simulation, the UAV manager is what keeps track of all the UAVs. There are four blocks in the main system that fall within the scope of this definition. The first two blocks are the UAV CRASH and UAV INTERCEPTED blocks, which serve similar functions. The first of these two blocks is the UAV CRASH block. This block uses a MATLAB s-function to determine if a UAV crosses the boundary of a no-fly zone. Though this should never happen with correct paths being assigned, the

function is still included for simulation completeness and is useful for error checking purposes.

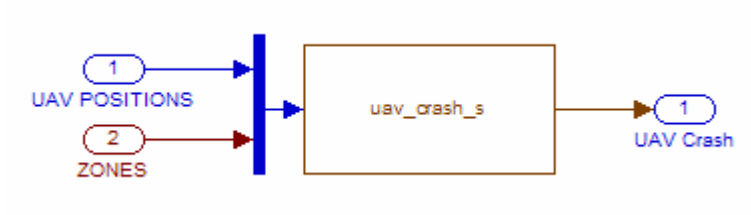

Figure 5.24: UAV CRASH block

The MATLAB function *uav_crash.m*, as found in Appendix B, uses the UAV positions as output by the aircraft dynamics and compares them with the no-fly zone information. If a UAV is determined to cross a boundary for a no-fly zone, the binary vector of UAV Crash is changed to a unit value for that UAVs position. That UAV is then deleted by the UAV DOWN block, which will be discussed shortly.

The second block is the UAV INTERCEPTED block. This block performs similarly to the UAV CRASH block. It uses a MATLAB s-function to compare the UAV positions with the threat positions and effective ranges. Figure 46 shows this block.

Figure 5.25: UAV INTERCEPTED block

If the function finds that a UAV has entered the effective range of a threat, the threat is simulated to have fired at the UAV. Note that each threat is considered to expend its entire armament when firing at a UAV. The amount of this armament is the same amount that was originally used to determine the probability-of-kill. For SAMs, a single missile determines this number, while for anti-aircraft artillery, the number of munitions

fired would be much higher. When a UAV is considered to have been fired upon, the simulator uses a random number generator to determine if the UAV got destroyed. For a random number between zero and one, if the number is less than the probability-of-kill for the threat, the UAV is considered destroyed and the binary vector UAV SHOT DOWN is changed to a unit value for that UAVs position. If the number is greater than or equal to the probability-of-kill, the UAV survives and continues on its path. Either way, the binary vector THREATS FIRED changes to a unit value for the firing threat and the Threats manager, discussed in Section 5.7, then removes that threat.

The third block that can be considered part of the UAVs manager is the UAV DOWN block. This block combines the two binary vectors UAV SHOT DOWN and UAV Crash into a single binary vector UAV DOWN that represents destroyed UAVs that are to be removed from the simulation.

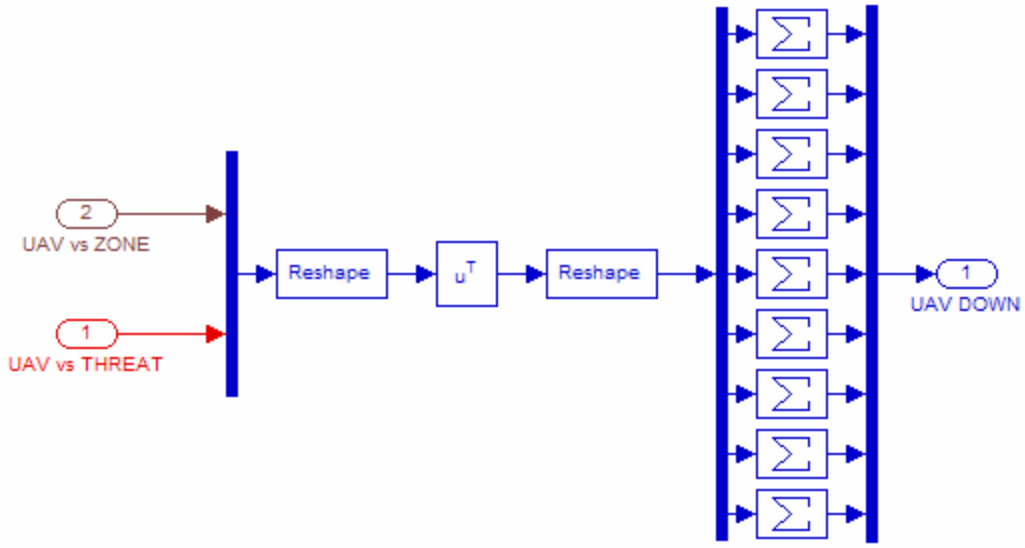

Figure 5.26: UAV DOWN block

The information from the UAV DOWN block is used in conjunction with the current UAV positions as output by the AIRCRAFT DYNAMICS block for the system entitled UAV MANAGER. The job of this system is to keep track of a current UAV matrix and to signal the path planning and task allocation scheme to replan if a UAV is lost. Figure 5.27illustrates the main subsystem.

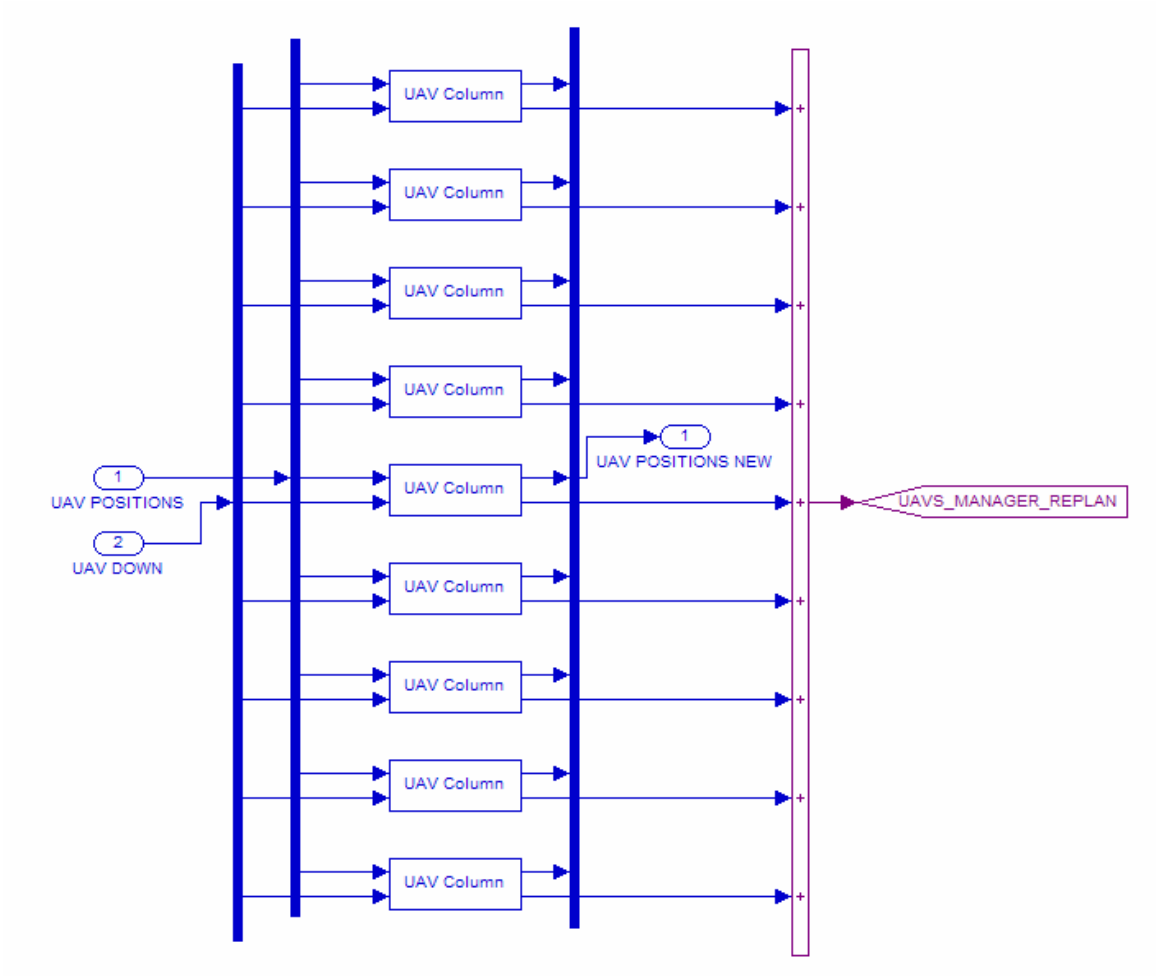

Figure 5.27: UAV MANAGER subsystem

This system is divided into a subsystem for each UAV that keeps track of the positions for each UAV, the velocity of the UAV, and if the UAV is destroyed or runs out of fuel. The binary value of the UAV DOWN vector associated with the individual UAV is combined with a binary value associated with the UAV running out of fuel to determine if the UAV is destroyed. The binary fuel value changes from zero to a unit value after a predetermined amount of time (for example, a LOCAAS type UAV has 30 minutes before it runs out of fuel). Changing the velocity of the aircraft to zero is used for a determination of UAV destruction. Because of inherent delays in the simulation, the change of velocity to zero is used to signal a replan as opposed to a binary value that is only a unit value for a single time step. Once the velocity changes to zero, the UAV is

officially removed from the list of UAVs and thus a replanning of the tasks and paths occurs only once for the loss of a UAV. For UAVs that are not used in the simulation, a zero vector is used to denote they do not exist. Because this vector is assigned at the start of the simulation and remains throughout, replanning is never based upon those UAVs.

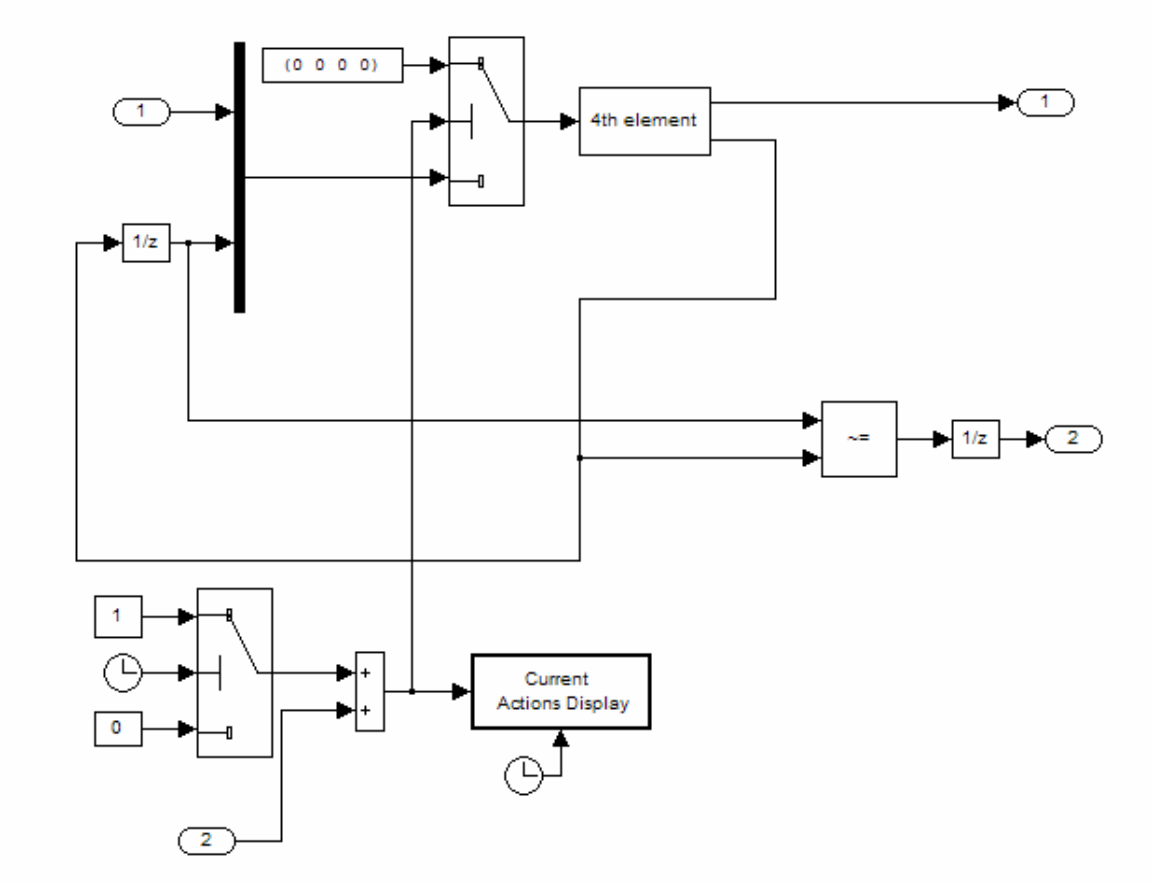

Figure 5.28: Individual UAV manager for tracking positions, velocity, and destruction

In addition to tracking UAV positions, velocities, and destruction, the individual manager has a subsystem to print a statement saying which UAV was destroyed and at what time. This statement is triggered when the combined binary number contains a unit value. The blocks to accomplish this function are seen in the next figure.

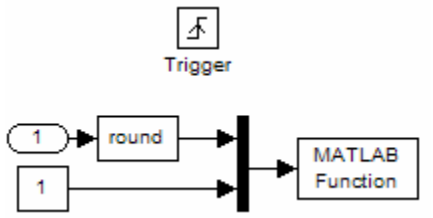

Figure 5.29: Printing blocks for UAV destruction

This concludes the UAVs manager description and the functions performed therein.

5.6 Targets Manager

The Target managing blocks keeps track of the state of each target. As assignments are completed by individual UAVs, target states change, and targets are eventually removed once confirmed as destroyed. There are two subsystems of the main system that performs the necessary management. The first subsystem is contained within the block TARGETS CLASSIFIER, while second is the TARGETS MANAGER.

The TARGETS CLASSIFIER has the job of tracking the states of each target. The five possible states of any given target are:

1. Indicated as a possible target
2. Identified as a target
3. Classified but not attacked
4. Attacked but not assessed
5. Assessed as destroyed

All targets start with the first state being assigned to them, where each is indicated as a possible target. The first assignment a UAV must do is to determine is the object really is a target. If the object is determined to be a target, then the second state is assigned stating

so. For objects determined to not be a target, a state indicating that it has been identified as not being a target is assigned. For targets determined to be such, the next possible state declares a target as classified but not attacked. UAVs must determine what type of target they are going to attack once the object is declared a target, but prior to the actual attack. Once a UAV attacks a target, that target receives the state 'attacked but not assessed'. The target must then be assessed as to whether the attack was successful or not. If so, the final state is assigned as 'assessed as destroyed'; otherwise, the target has not been successfully destroyed and must be reattacked. This is accomplished by returning the target to state 3, indicating that the target has been classified but not attacked. The target is then reattacked and reassessed.

The subsystem performing this state management is shown in Figure 5.30.

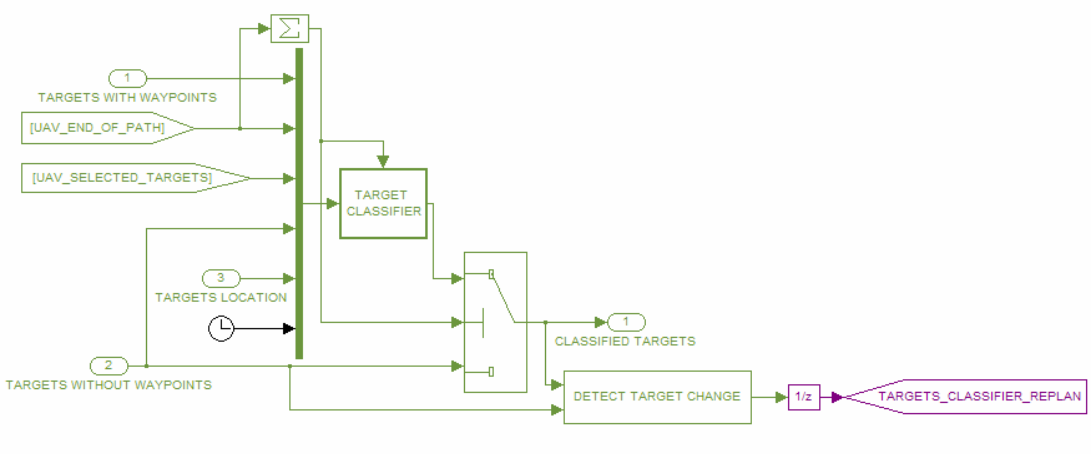

Figure 5.30: Target State Manager

This manager features two parts. The first part uses a MATLAB s-function called *target_classifier_s.m* to perform the classification task. This function can be viewed in Appendix B. Individual UAVs signal when they have reached their assigned target. Whenever this occurs, this manager increases the state of the target for successful state succession, and removes objects that are found to be not actual targets and targets that are assessed as destroyed. However, for simulation purposes, it also includes random

probability that objects are not targets and that targets will take more than one attack for successful destruction. Figure 5.31 contains the function used for classifying purposes.

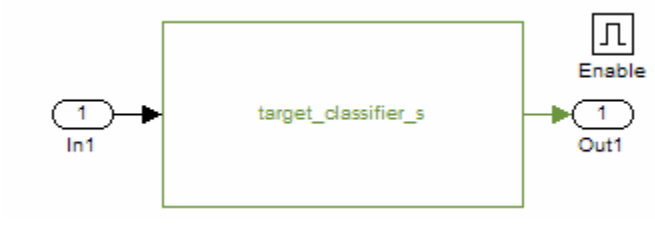

Figure 5.31: Target classifier function

The second part of this subsystem is used to signal replanning to occur. Whenever a target changes states, a new task must be performed. This task must go through the path planning and task allocation scheme to be assigned to an individual UAV, so thus a signal is issued to cause a replan. Figure 5.32 shows how an inequality between the former states of all targets and the new states of the targets is used to enable a replan.

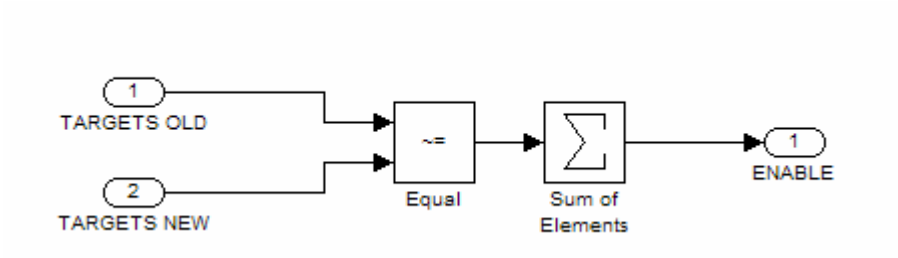

Figure 5.32: Part of target classification used for signaling replan

The second subsystem considered to be part of the managing of targets is the block called TARGETS MANAGER. This subsystem handles the tasks of tracking pop-up targets and issuing replanning commands based upon new target information. The following figure is of the blocks used for this purpose.

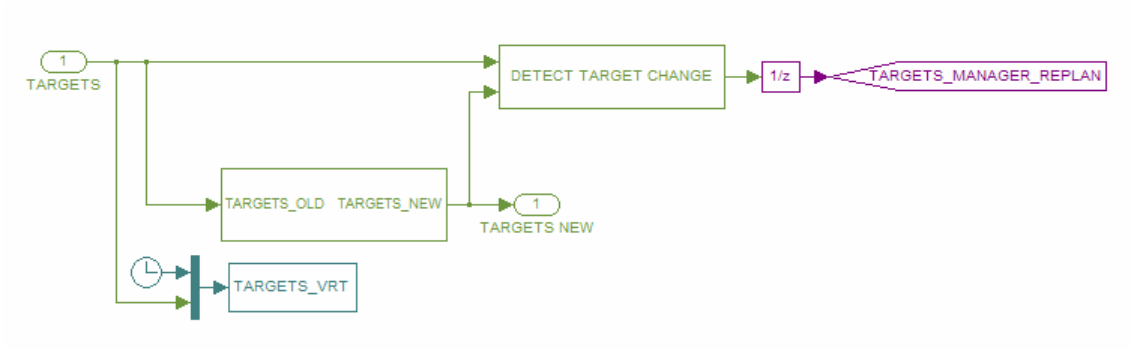

Figure 5.33: TARGETS MANAGER

This subsystem contains two smaller systems within itself. The first of these systems is identical to the one shown in Figure 5.32. This system uses a comparison of old target information and current target information to determine when a change has occurred. When a change occurs, a signal is sent to initiate a replan.

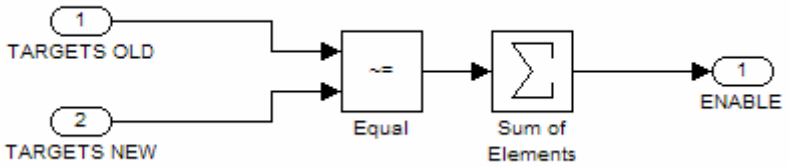

Figure 5.34: Part of target management used for signaling replan

The second, small system within the TARGETS MANAGER system is used for managing pop-up targets. Pop-up targets have been declared by the omniscient user to show up on the list of targets at a predetermined time. This manager tracks the time, and at the predetermined time, the target is included into the target matrix. Figure 5.35 shows the nine possible targets that can be used with an associated pop-up time.

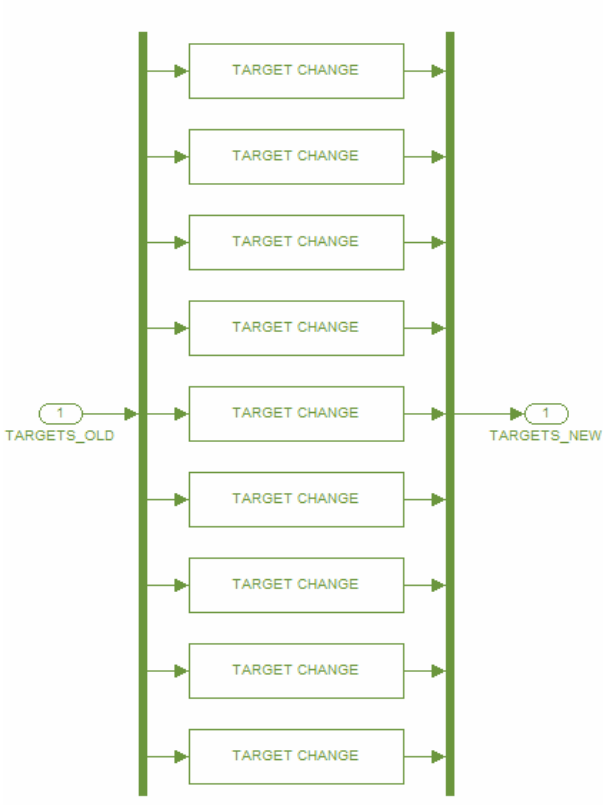

Figure 5.35: Pop-up target manager

Under each block labeled TARGET CHANGE lies the blocks shown in Figure 5.36. These blocks control the pop-up function for each individual target and display to the user whenever the pop-up occurs.

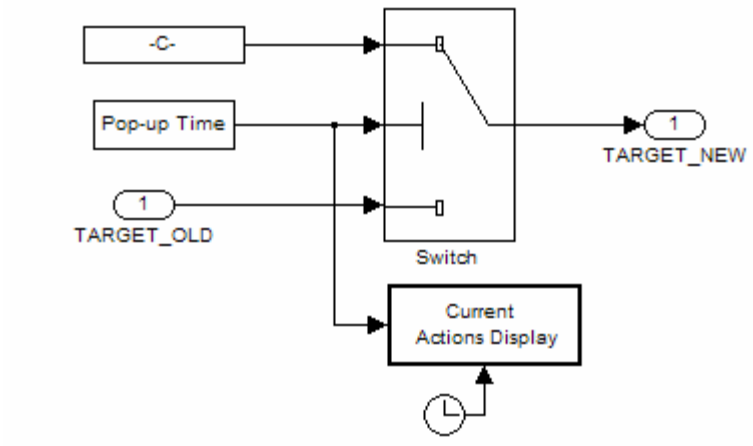

Figure 5.36: Pop-up target manager for an individual target

5.7 Threats Manager

Aside from the state change functions of the targets manager, the threats manager is quite similar to the targets manager. The THREATS MANAGER is shown in the following figure. As with the TARGETS MANAGER subsystem, there are two parts used to control the replan signal and the new list of threats.

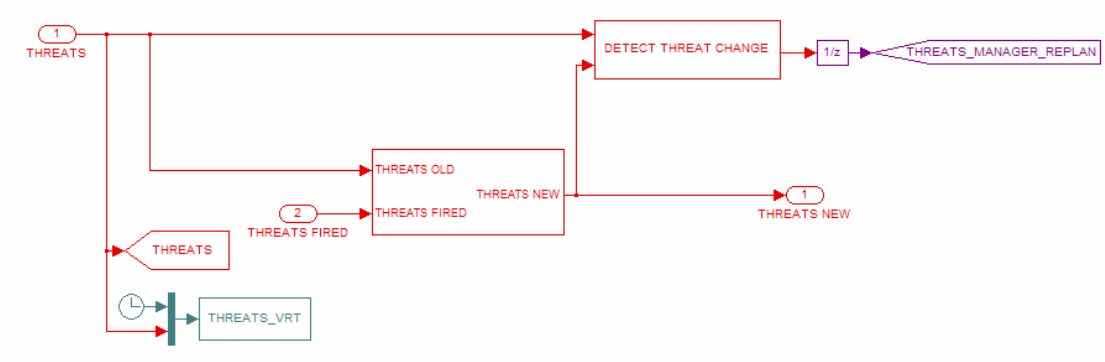

Figure 5.37: THREATS MANAGER

The first part controls the replan initialization. This part is the same as the one used in the targets manager, as shown in Figure 5.34. This part compares the list of old threats to the current list of threats. If a change is detected, such as a new pop-up threat being added or an old threat firing and then being removed, the replan signal is issued.

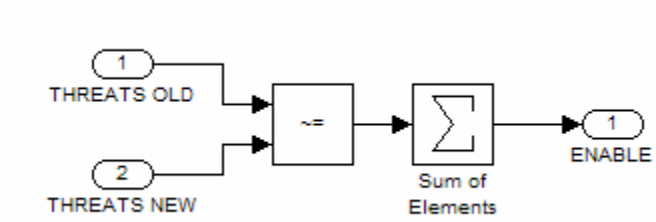

Figure 5.38: Part of threat management used for signaling replan

The second part of the THREATS MANAGER contains a set of 15 THREAT CHANGE blocks, as shown in Figure 5.39.

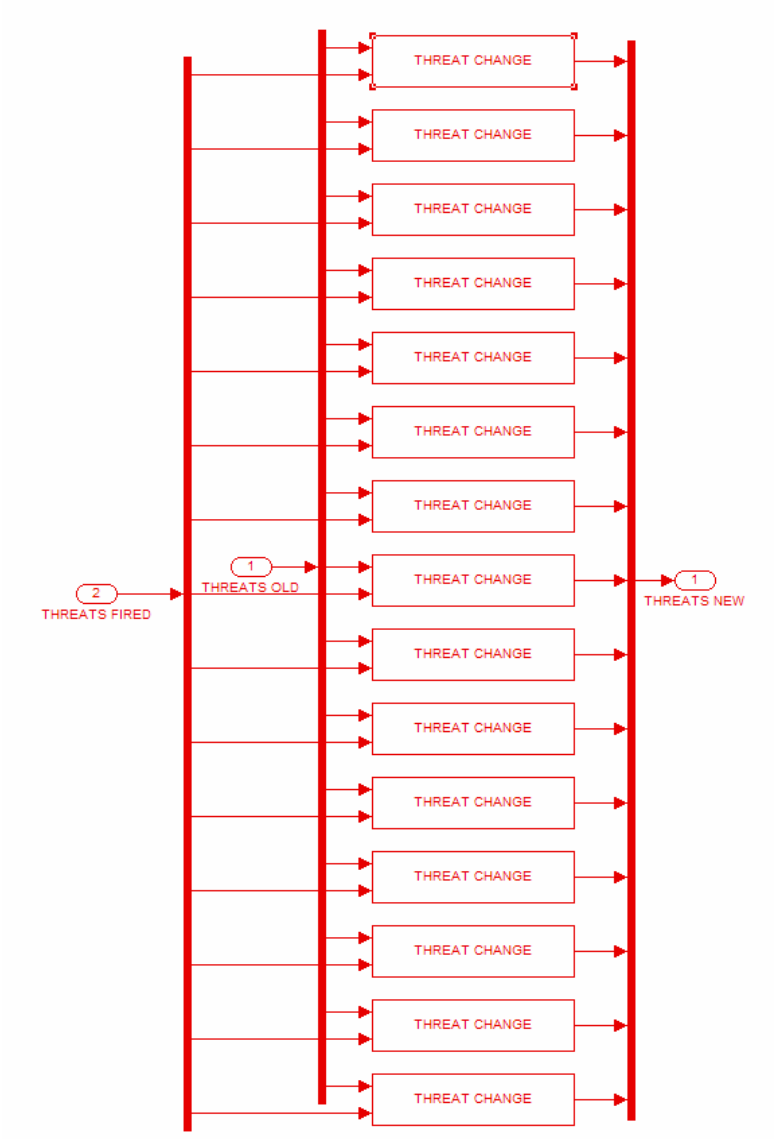

Figure 5.39: THREAT CHANGE blocks

These blocks each contain a subsystem that controls the pop-up function for each individual target and displays to the user whenever the pop-up occurs. In addition to these functions, this subsystem also tracks if and when the threat fires. If a threat is determined to have fired as declared by the UAV SHOT DOWN system, the threat is removed from the list of threats, as explained in section 5.5

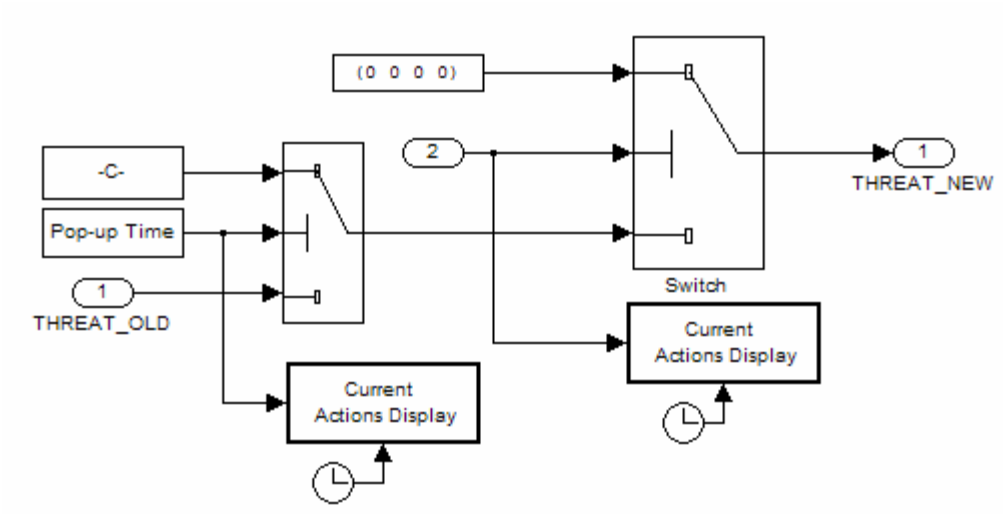

Figure 5.40: Pop-up and firing threat manager for an individual threat

5.8 Simulation Outputs

The outputs of this simulation are threefold. The first is output to the MATLAB command window. This output initially displays all inputted information to the user. This information includes UAV locations, altitudes, and velocities, target locations and initial states, threat locations, ranges, and probability-of-kill, and no-fly zone coordinates and radii. After this initial display, the command window output displays whenever a replan occurs, at what time, and what event caused it. The second types of output are static plots showing the planned paths and allocated tasks. These plots can be turned on or off, and when on, are displayed every time a replan is performed. The last output is a graphical visualization using moving plots to illustrate the simulation.

The first two simulation outputs are illustrated through an example. This example is relatively simple, to keep the length down for necessary plots to shown simulation steps. This simulation consists of four UAVs, three static targets, a single pop-up target occurring at 100 seconds, three no-fly zones of radius nine kilometers, two static threats, and one pop-up threat appearing after 150 seconds. Figure 5.41 illustrates the initial battlefield setup. Note that the scales along the axes are in terms of kilometers. The

UAVs are shown as blue diamonds numbered 1 through 4 along the left side of the battlefield. The static targets are green ‘x’s, while the single pop-up target is shown as a green ‘+’. The no-fly zones are the obvious black circles. Threats are shown as a red star with surrounding effective radius for the static variety, and the pop-up threat is the large read range with the red ‘O’ at the center.

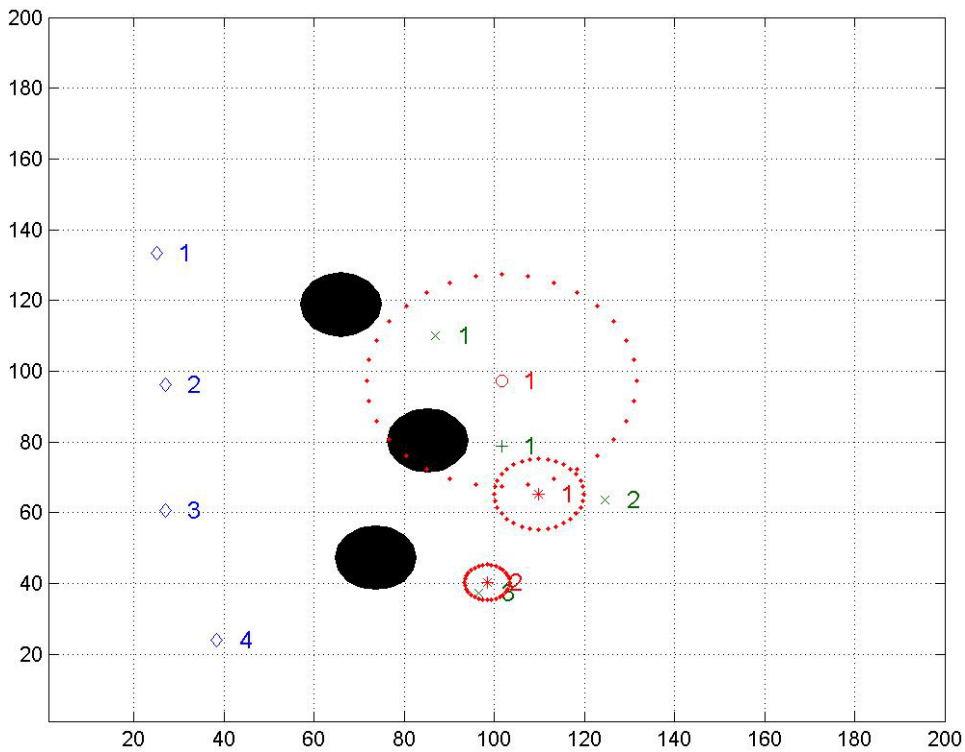

Figure 5.41: Initial battlefield setup

The first outputs when the simulation is started are the following expressions printed in the MATLAB command window:

UAV 1 exists at location 25 x, location 133 y, altitude 2 km, and is flying at 130 m/s.

UAV 2 exists at location 27 x, location 96 y, altitude 2 km, and is flying at 130 m/s.

UAV 3 exists at location 27 x, location 61 y, altitude 2 km, and is flying at 130 m/s.

UAV 4 exists at location 38 x, location 24 y, altitude 2 km, and is flying at 130 m/s.

Target 1 indicated to be at location 87 x, location 110 y , and with an estimated value of 40.

Target 2 indicated to be at location 125 x, location 64 y, and with an estimated value of 70.

Target 3 indicated to be at location 97 x, location 37 y, and with an estimated value of 100.

No-Fly Zone 1 exists at location 66 x, location 119 y, and with a radius of 9 km.

No-Fly Zone 2 exists at location 85 x, location 80 y, and with a radius of 9 km.

No-Fly Zone 3 exists at location 74 x, location 47 y, and with a radius of 9 km.

Threat 1 exists at location 110 x, location 65 y, with a range of 10 km, and has a probability of kill of 80%.

Threat 2 exists at location 98 x, location 40 y, with a range of 5 km, and has a probability of kill of 50%.

These expressions completely specify the initial battlefield setup in words. From here out, the example will proceed with text stating what event occurred, and a figure illustrating the path planning and task allocation based on the new information will immediately follow.

Path Planning ran at time 0.

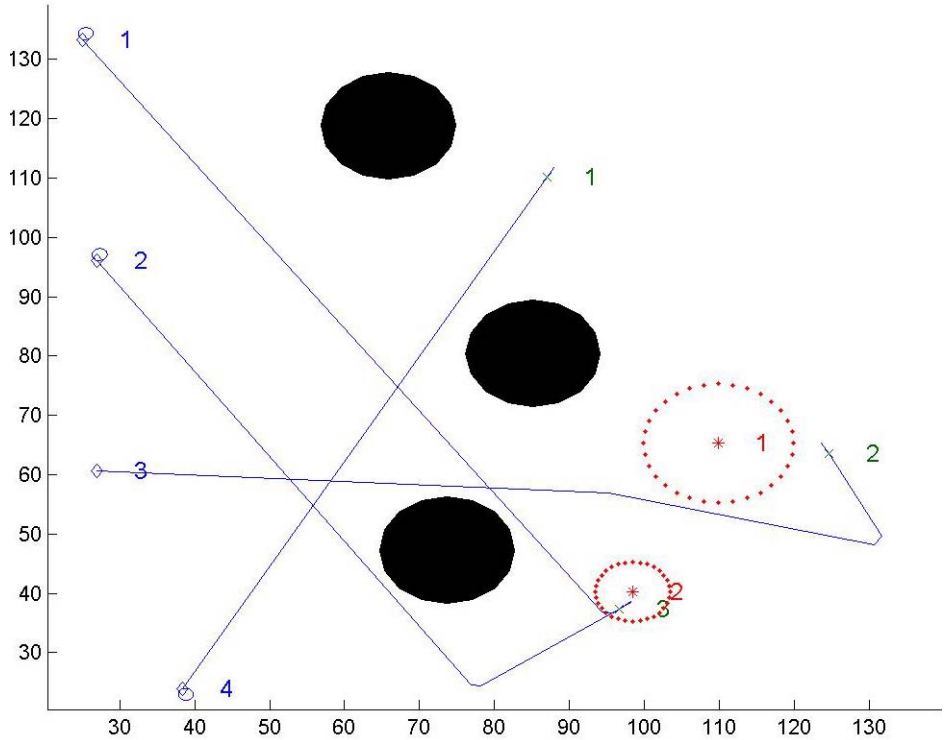

Figure 5.42: Path Planning and Task Allocation occurring at time 0

Target 4 has popped up at time 100.

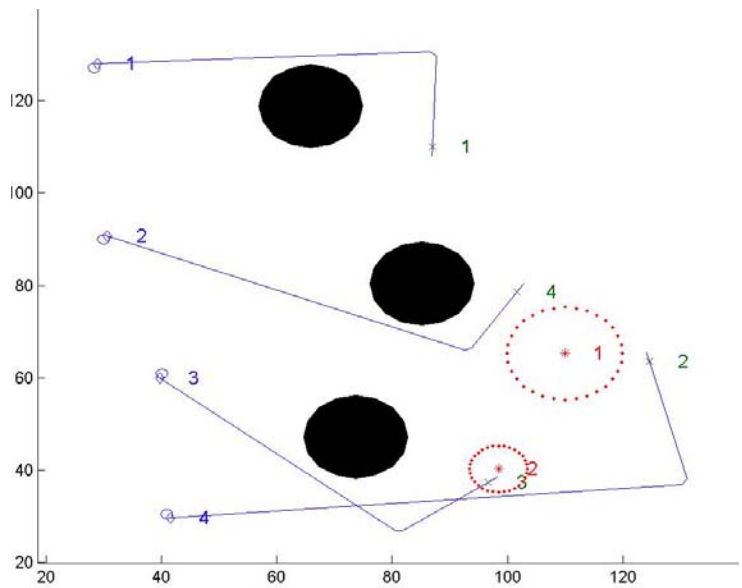

Figure 5.43: Path Planning and Task Allocation occurring at time 100

Threat 3 has popped up at time 150.

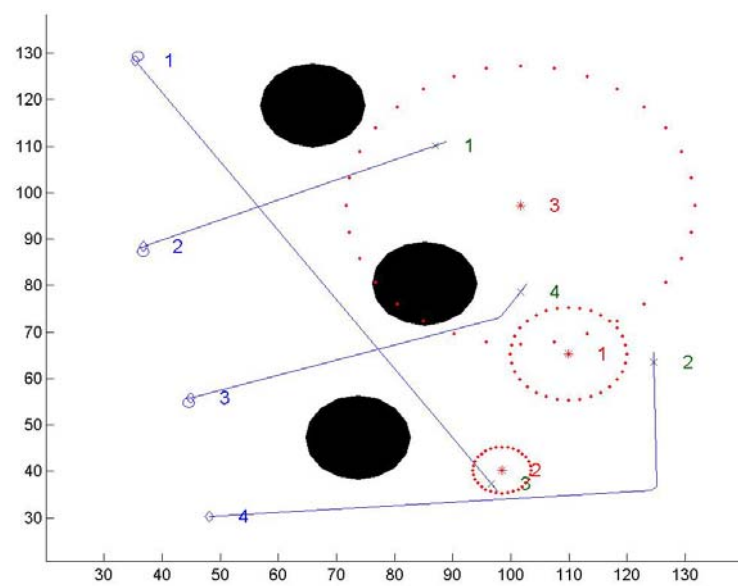

Figure 5.44: Path Planning and Task Allocation occurring at time 150

Threat 3 has fired at time 325.

UAV 2 has been destroyed at time 325. .

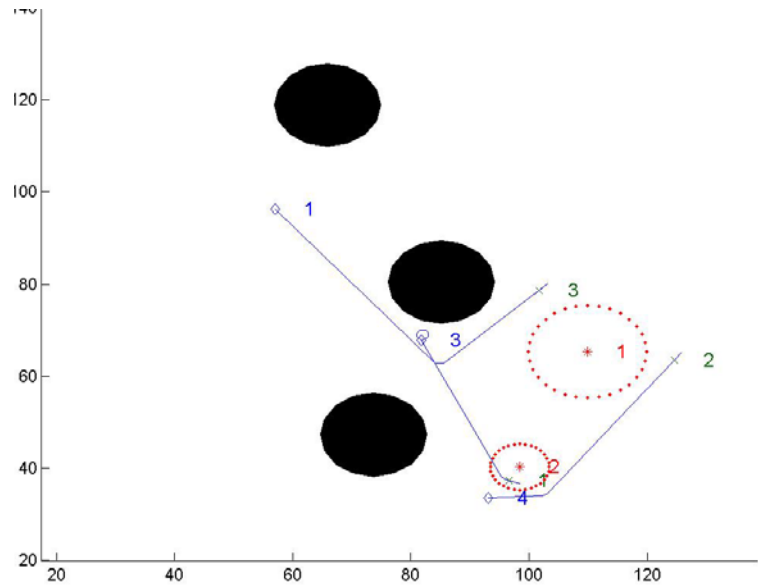

Figure 5.45: Path Planning and Task Allocation occurring at time 325

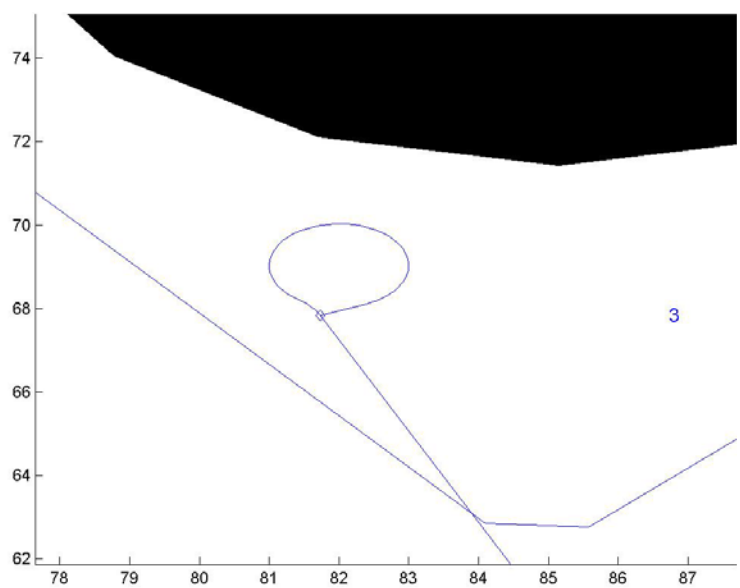

Figure 5.46: Detail of UAV 3 turning to now attack target 1 at time 325

Threat 2 has fired at time 462.

UAV 3 has been destroyed at time 462.

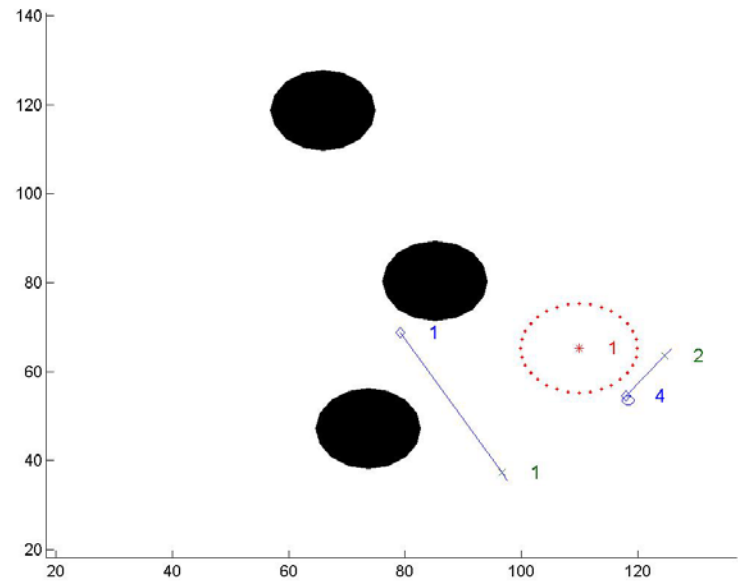

Figure 5.47: Path Planning and Task Allocation occurring at time 462

Target 2 (value 70) identified as NOT a target at time 538 by UAV 4.

Target 2 has been removed from target status at time 538.

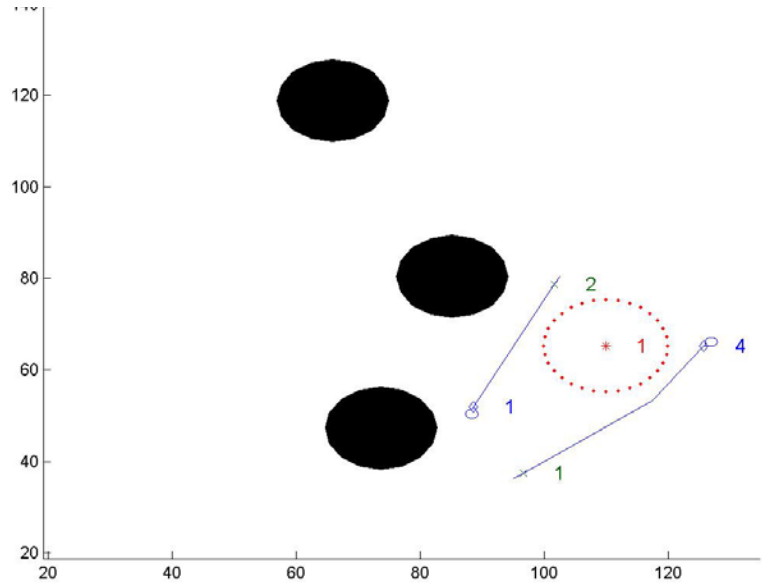

Figure 5.48: Path Planning and Task Allocation occurring at time 538

Target 4 (value 50) identified as a target at time 688 by UAV 1.

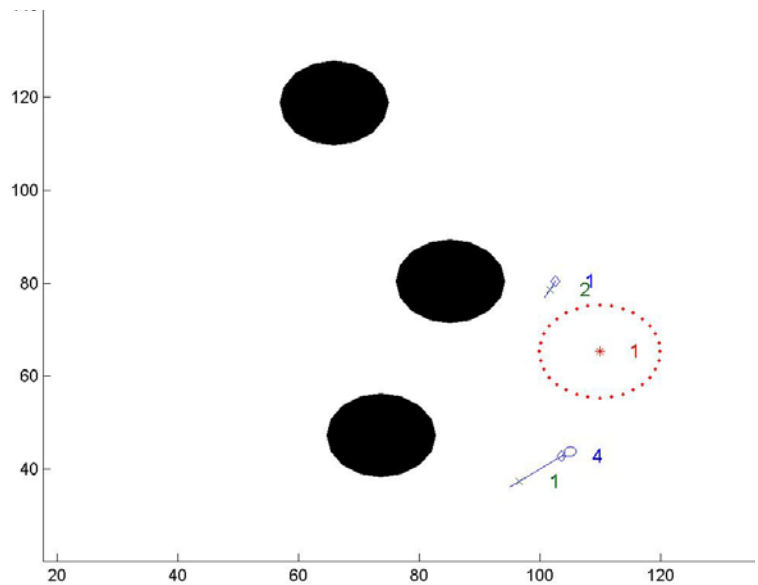

Figure 5.49: Path Planning and Task Allocation occurring at time 688

Target 4 (value 50) classified not attacked at time 704 by UAV 1.

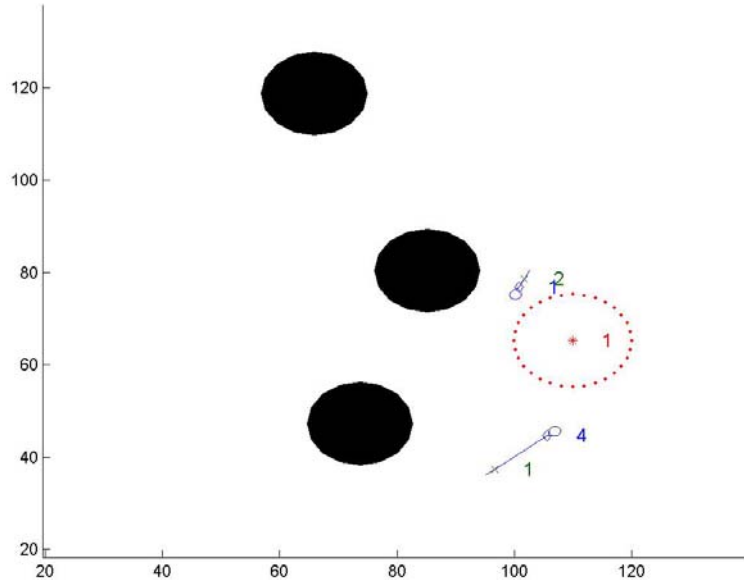

Figure 5.50: Path Planning and Task Allocation occurring at time 704

Target 4 (value 50) attacked not assessed at time 749 by UAV 1.

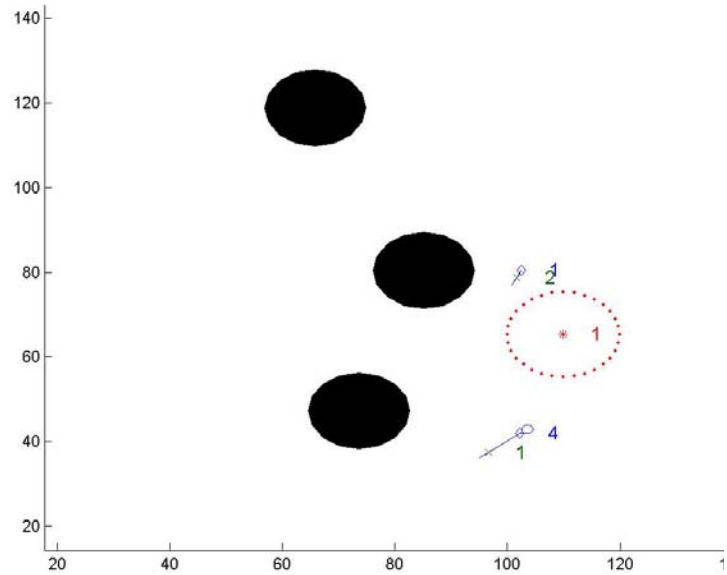

Figure 5.51: Path Planning and Task Allocation occurring at time 749

Target 4 (value 0) assessed as destroyed at time 764 by UAV 1.

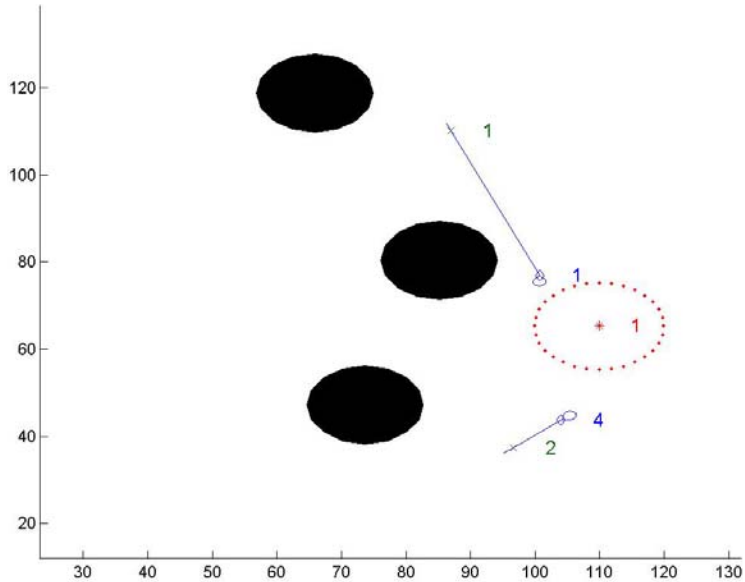

Figure 5.52: Path Planning and Task Allocation occurring at time 764

Target 3 (value 100) identified as a target at time 838 by UAV 4.

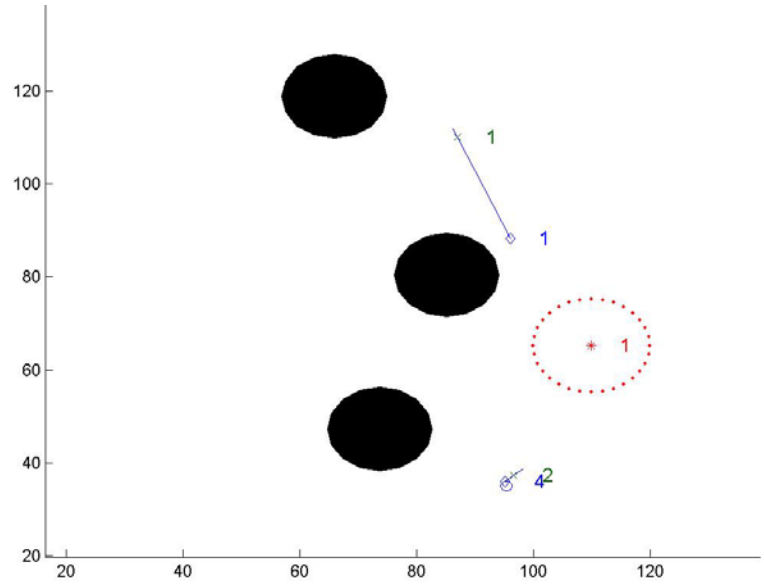

Figure 5.53: Path Planning and Task Allocation occurring at time 838

Target 3 (value 100) classified not attacked at time 878 by UAV 4.

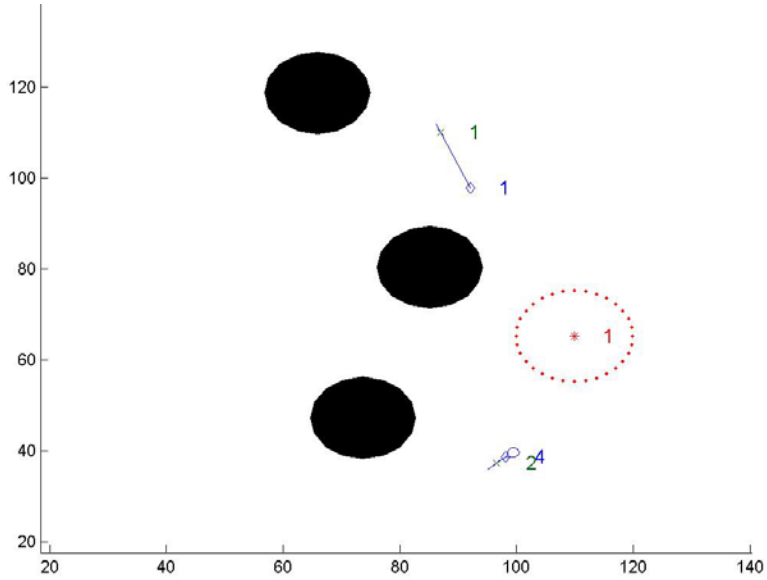

Figure 5.54: Path Planning and Task Allocation occurring at time 878

Target 3 (value 100) attacked not assessed at time 921 by UAV 4.

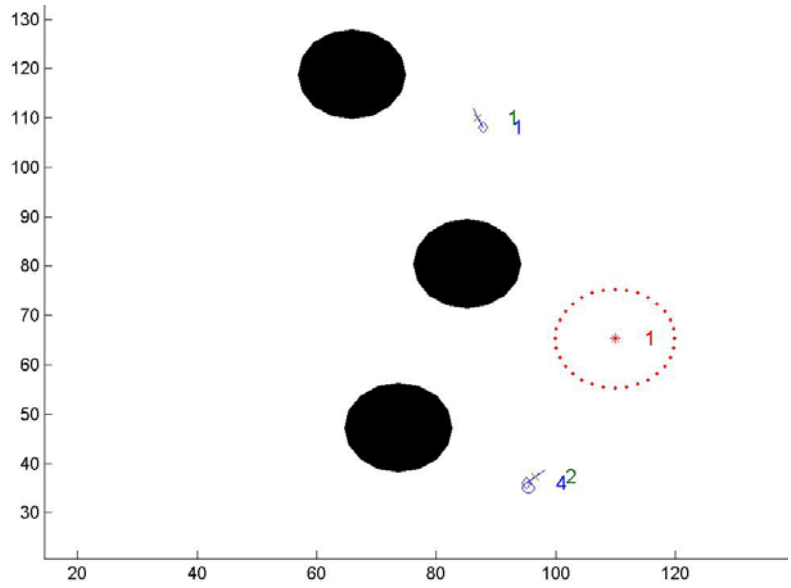

Figure 5.55: Path Planning and Task Allocation occurring at time 921

Target 1 (value 40) identified as a target at time 938 by UAV 1.

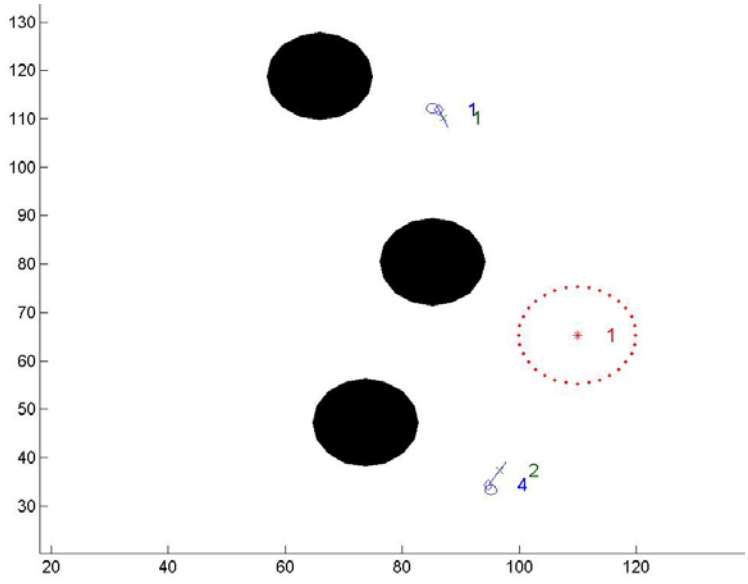

Figure 5.56: Path Planning and Task Allocation occurring at time 938

Target 1 (value 40) classified not attacked at time 978 by UAV 1.

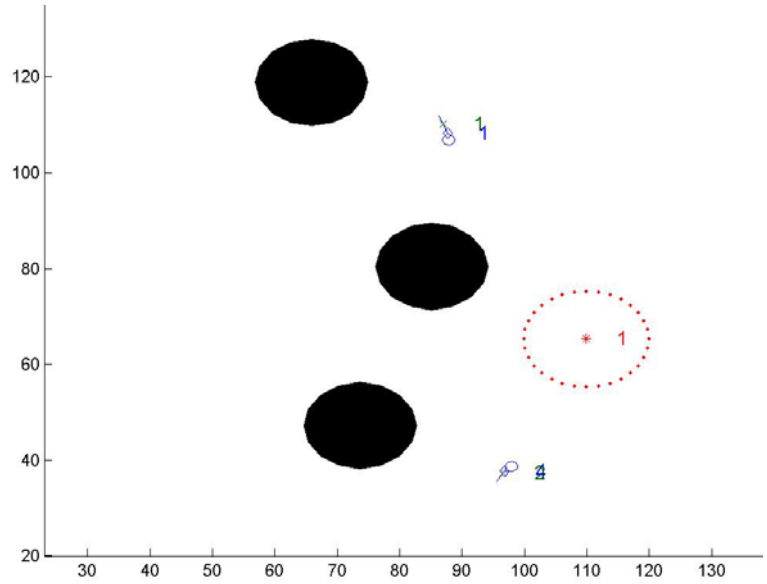

Figure 5.57: Path Planning and Task Allocation occurring at time 978

Target 3 (value 0) assessed as destroyed at time 1014 by UAV 4.

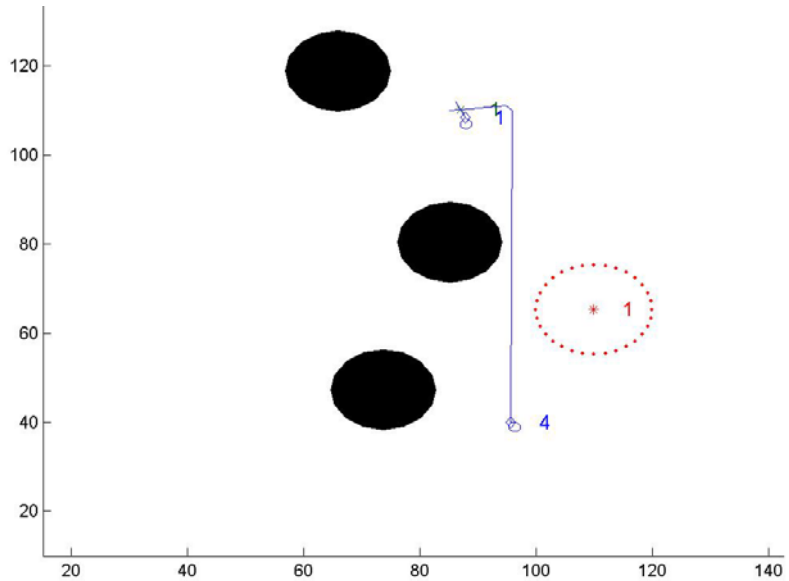

Figure 5.58: Path Planning and Task Allocation occurring at time 1014

Target 1 (value 40) attacked not assessed at time 1056 by UAV 1.

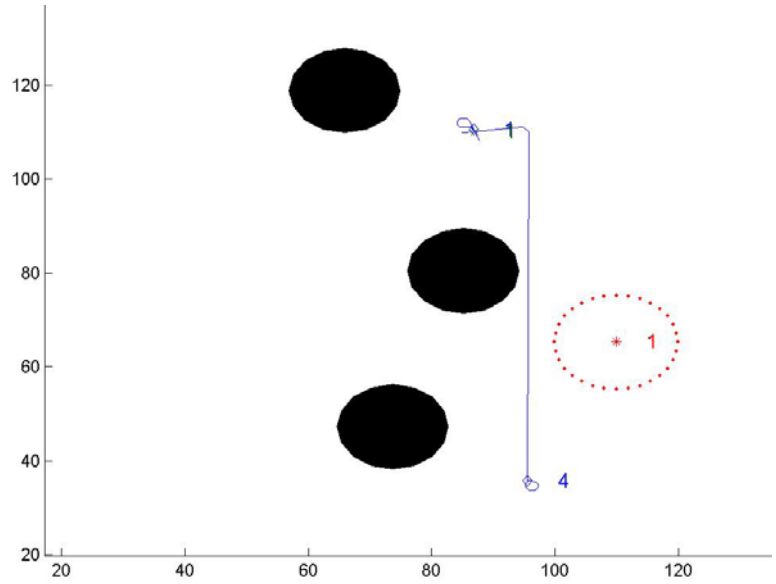

Figure 5.59: Path Planning and Task Allocation occurring at time 1056

Target 1 (value 0) assessed as destroyed at time 1098 by UAV 1.

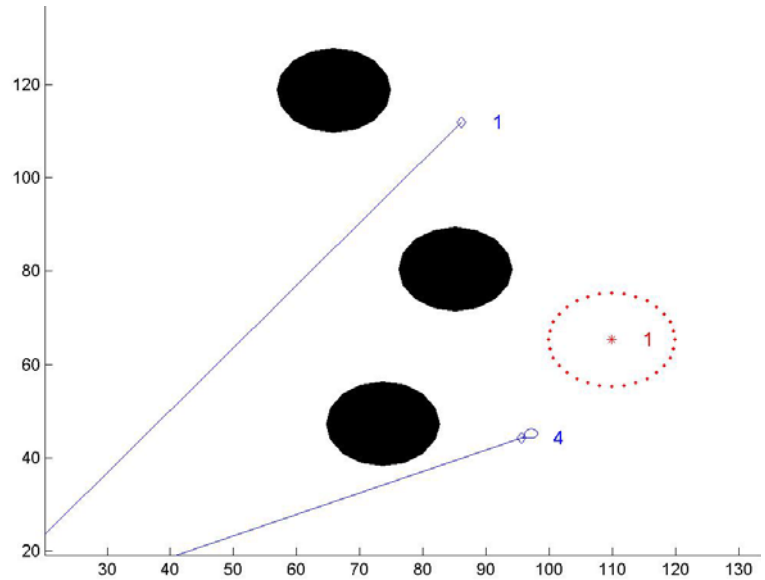

Figure 5.60: Path Planning and Task Allocation occurring at time 1098

Since no more tasks are to be allocated, all UAVs are assigned to return to a predetermined set of home-base coordinates (typically the origin is used for simulation). It should be noted that the static plots presented here are based off of the *plot_uav.m* MATLAB code shown in Appendix A. Since is uses the knowledge presented by the path planning and task allocation scheme, there is an occasional renumbering of targets shown on the static plots. However, the actual numbering kept by the targets manager is the same as the original numbering, even as targets are removed from the list. The MATLAB command window printouts are also based upon this list, rather than the localized renumber of the path planning and task allocation scheme.

The simulation presented in this chapter has been a centralized version that fulfils the third and fourth research objectives. This simulation has been designed to simulate a maximum of nine UAVs, nine targets, fifteen no-fly zones, and fifteen threats, and encompasses time-varying simulation aspects, such as UAVs being destroyed, targets and threats popping-up at a time unknown to the UAVs, and simulates accurate battle management.

Chapter 6

Decentralized Path Planning and Task Allocation

6.1 Main Simulation System

The decentralized simulation developed here is a truly decentralized control scheme for a team of UAVs. This approach is an extension of the centralized version discussed in the preceding chapter. The following figure illustrates the new simulation with a maximum of nine UAVs and corresponding communications between each.

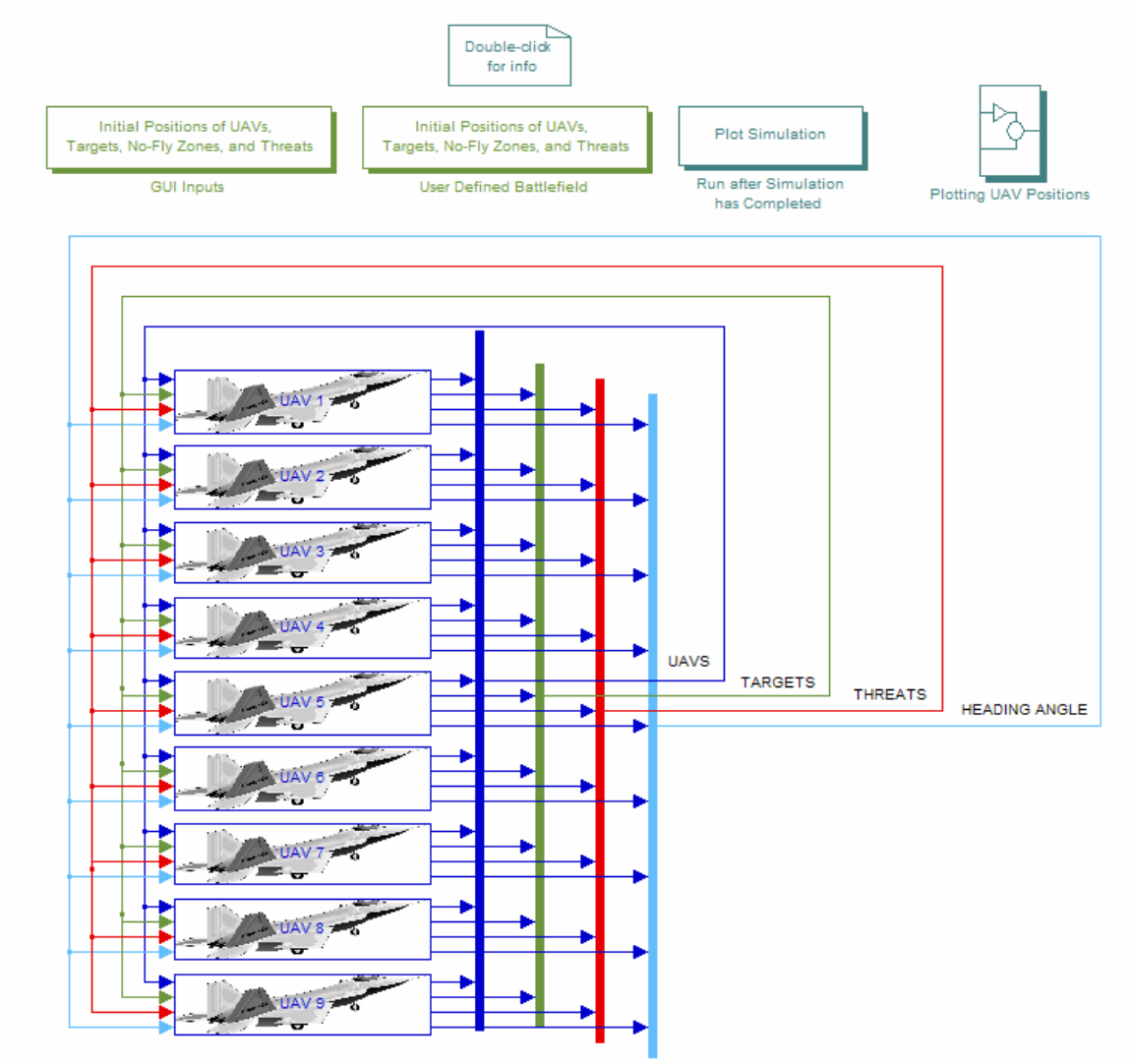

Figure 6.1: Main simulation system for decentralized UAV control

As seen with the main system, this scheme has no center controller or even leader. All UAVs are used to make decisions and perform tasks. The theory behind this decentralized approach is the following statement: *a team of UAVs with every member possessing full situational awareness (SA) will always arrive at the same correct decision.*

6.2 Individual UAV System

The theoretical statement made in the last section has been applied to designing an individual UAV system that makes decisions for that UAV and performs similar management as the centralized simulation. Each UAV uses the same path planning and task allocation scheme as the centralized version discussed but then uses only the information necessary for that UAV to perform its allocated task. Figure 6.2 contains the main system that is used within each individual UAV. The similarities between the centralized simulation and the system used for individual UAVs should be noted. The differences between these systems will be discussed shortly.

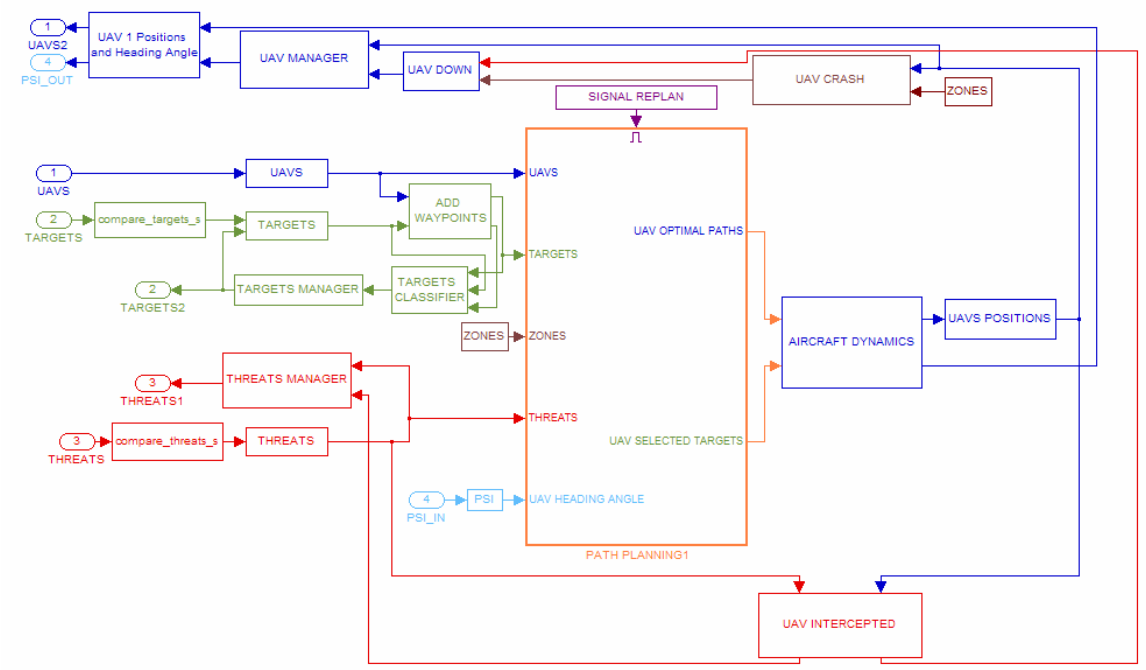

Figure 6.2: Main system for individual UAVs

Essentially each UAV in the team is running the above system. Necessary information is passed between all cooperating UAVs. That information is used by each individual UAV to run a path planning and task allocation scheme. Because the information communicated between UAVs is current and globally known, each UAV is able to run the path planning and task allocation scheme and arrive at the same decisions as every other UAV. This minimizes the amount of information communicated between UAVs and eliminates the need to have a central path planning and task allocation scheme issuing commands to each UAV; however, it implies the need for availability of substantial computational power for the on-board computer of each UAV. Each individual UAV uses their planned path to perform its assigned task. Because each UAV has arrived at the exact same path planning and task allocation assignments, the decentralized scheme progresses much like the centralized version.

6.3 UAV Communications

There are three main pieces of information that need to be communicated: information about individual UAVs, updated target information, and updated threat information. Since the no-fly zones are stationary (scenarios including 'pop-up mountains' are unrealistic), this information does not need to be communicated by the individual UAVs. As can be seen in the first figure of this chapter, there are four outputs of each UAV: the first output is the positions of the individual UAVs. Second is the individual UAV's knowledge of the targets. Third is the UAV's knowledge of the threats, and lastly is the individual UAV's current heading angle. Each of the outputs of the individual UAVs is multiplexed and sent to every UAV. This is the information that each UAV is communicating with every other UAV.

Once this information enters the individual UAV, several things happen to correctly process this information. For the UAV positions and heading angle, the information can be used directly, since there is only one set of information about UAV

‘X’, because only UAV ‘X’ output that information. For the targets, the UAV has the knowledge of each other UAV for the targets. The individual UAV then compares that knowledge to what it already knew about those targets. For example, if all but one UAV indicates that target ‘Y’ has been attacked but battle damage assessment has not been accomplished, and the one dissenting UAV indicates that battle damage assessment has been performed (because he performed it), then each of the individual UAVs update their information to indicate that target ‘Y’ has had battle damage assessment performed, and a replan occurs in each of the UAVS. The same occurs with the inputs for the threats. If a threat fires, and a single UAV indicates that it fires, then all the UAVs will update their information showing that that threat fired. These comparisons are accomplished using the MATLAB code *compare_targets.m* and *compare_threats.m*, which can be found in Appendix B.

This information will give all UAVs full situational awareness and the ability to correctly make planning decisions. The next chapter will investigate issues occurring whenever all team members do not possess full situational awareness, which indicates the above information is not being properly communicated.

6.4 Individual UAV Calculations

As aforementioned, the system running inside each individual UAV shares many features with the centralized simulation, but also contains a number of differences to allow the decentralized simulation to occur. The steps to the path planning and task allocation scheme essentially remain the same. In the centralized simulation, this entailed using the most current information, running an ADD WAYPOINTS subsystem to generate extra waypoints or suspend lower-valued targets for the team, and finally perform the path planning and task allocation process. The decentralized version remains the same, with current information (now coming from team communication) and waypoint information being used for path planning and task allocation.

The first difference occurs with the AIRCRAFT DYNAMICS subsystem. In the centralized version, the central processor simulated the dynamics of all UAV team members. For individual UAVs in the decentralized version, there is no need to simulate dynamics for other UAVs that an individual UAV certain does not control. The figure below shows the subsystem of the UAV DYNAMICS for the first UAV.

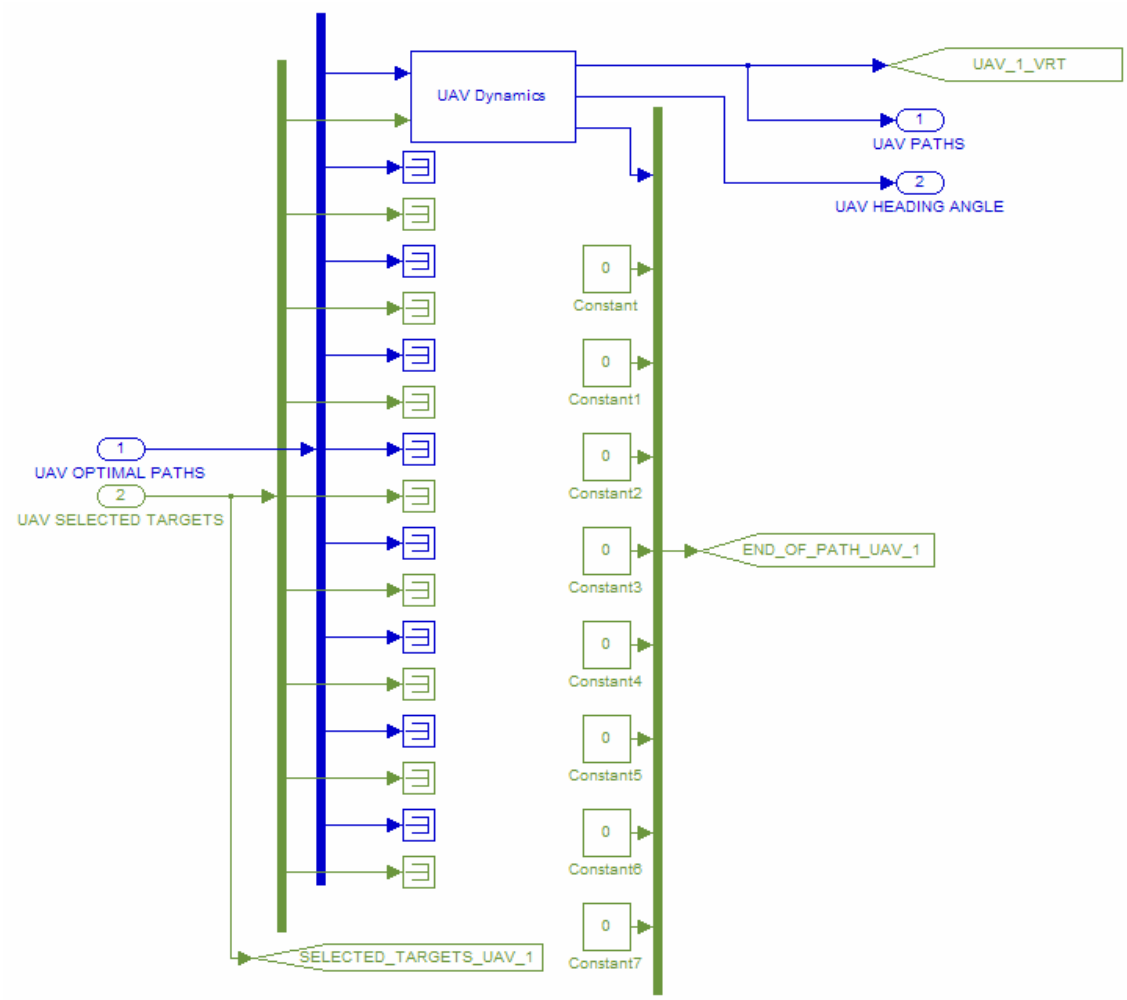

Figure 6.3: 'UAV Dynamics' blocks for UAV 1

The path planning and task allocation scheme generates a task assignment and path for the individual UAV along with the predicted assignments of all other UAV team members. For correct communications, all UAVs will know what every other UAV will be doing. As mentioned with the dynamics, there is no need after the path planning and

task allocation assignment for an individual UAV to be concerned with other UAVs who it certainly does not control. The aircraft dynamics for the individual aircraft are found using the exact same approach as described in Section 5.4. Once the actual aircraft positions and rotations are found using the aircraft model, the UAV Positions block passes on the positions of that UAV, as seen in Figure 6.4.

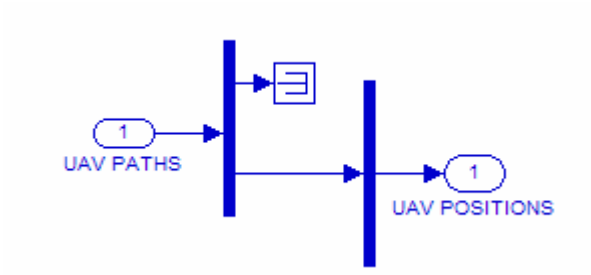

Figure 6.4: UAV Positions block

The position of the UAV is then used in the same way as the central version to determine if the individual UAV passes within the boundary of any threat or no-fly zone. The calculations are much simpler here since only a single UAV position is compared with known threat and no-fly zone positions. The former UAV DOWN vector is turned into a single binary number to signal UAV loss. This information is then used in conjunction with the position to signal the group of the loss of the individual UAV, as performed in the UAV MANAGER, shown in Figure 6.5.

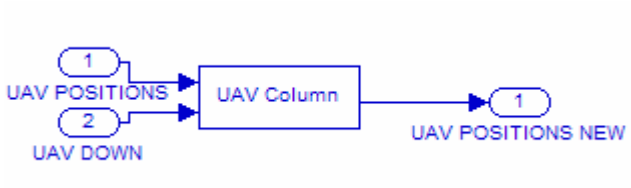

Figure 6.5: Individual UAV MANAGER subsystem

The UAV no longer issues replanning signals itself. To ensure the entire UAV team replans as the new surviving UAV information becomes available, the replan has been relocated to the initialization block for the UAVs, as shown in the next figure.

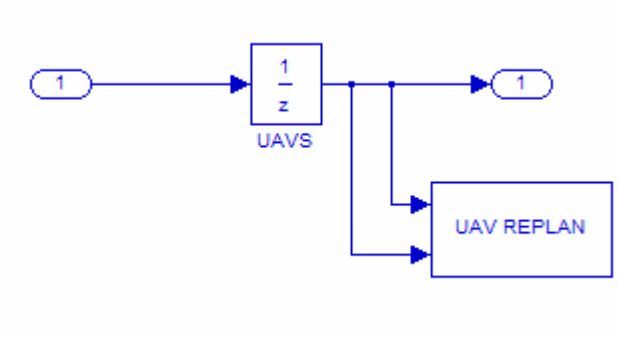

Figure 6.6: UAV initialization block with UAV REPLAN subsystem

This block still serves its initialization function uninterrupted, as seen in the upper branch of the system, but has the addition of the UAV REPLAN block. This subsystem compares the UAVs current knowledge of the UAV team with its former knowledge of the UAV team. When a difference is detected that indicates a loss of one or more members, the replan signal is issued. Figure 6.7 illustrates the new UAV REPLAN subsystem.

Figure 6.7: UAV REPLAN subsystem

Whenever the other UAVs become aware of the loss of a member, each UAV replans based on the surviving UAV positions and current target and threat information. Each UAV contains the same target and threat management that the central version contains. Each UAV has knowledge of every threat and target and the corresponding states. As mentioned, whenever any UAV presents new information to the group about a threat or target, all team members update their information and each replans accordingly.

Target management is conducted in a similar manner. The new TARGETS MANAGER still determines if a new target is added to the list of current targets, but the replan signal for target changes (including target state changes) occurs within the TARGETS initialization block, as seen in Figure 6.9, which is preceded by the figure of the new manager.

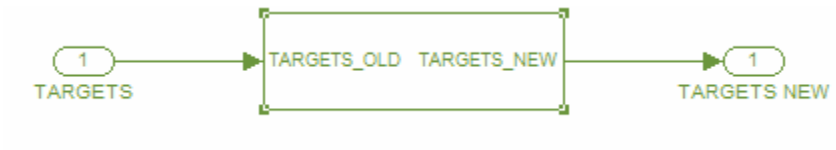

Figure 6.8: TARGETS MANAGER

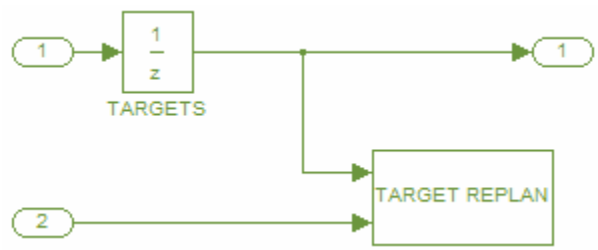

Figure 6.9: TARGETS initialization block with UAV REPLAN subsystem

The TARGETS REPLAN subsystem functions the same as the UAV REPLAN system. This system detects changes in the same manner as the comparing system originally described in Section 5.6 in Figure 5.32.

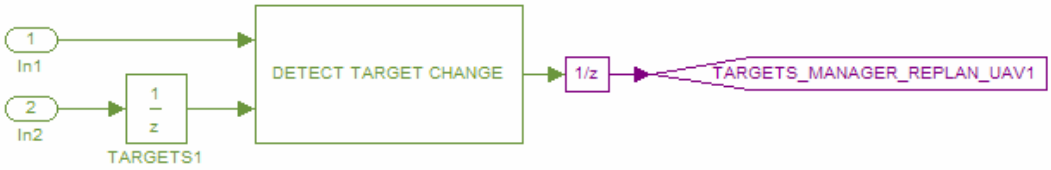

Figure 6.10: TARGET REPLAN subsystem

6.5 Simulation Outputs

The outputs of this simulation are the same as the centralized version. As with the centralized simulation, there are three outputs; however, only two of them would be typically used with a decentralized simulation. The first is, again, the output to the MATLAB command window. Initially, it displays the UAV locations, altitudes, and velocities, target locations and initial states, threat locations, ranges, and probability-of-kill, and no-fly zone coordinates and radii. After this, occurring events will be displayed by the UAV that detected them, and each UAV will display whenever it replans. Typically, this means that an event will happen, and then there will be nine displays for replanning. There can be a maximum of nine UAVs for this simulation, and while less than the maximum can be ran, the path planning and task allocation scheme still runs in the nonexistent UAVs, even though they never receiver or perform tasks.

The second types of output are static plots showing the planned paths and allocated tasks. These plots typically are not used with a decentralized scheme. Because the path planning and task allocation scheme actually produces the plots, there will be nine sets of plots for each occurring event. Also, these plots display only what each individual UAV knows, not what could realistically be happening. Contrasting information such as misinformation or loss of information will produce different plots based upon what each UAV sees as the correct information. This could be helpful in situations where the user wants to find out 'who-knows-what', but generally these plots would not be of much use. Misinformation occurrences are discussed in the next section.

The last output is a graphical visualization using moving plots to illustrate the simulation. Coupled with a statement of the events occurring in the MATLAB command window, this moving plot greatly helps the user to visualize the simulation. A short example to illustrate this plotting is shown in the form of captured images. Figure 6.11 shows an initial battlefield setup with four UAVs, three targets, three no-fly zones, and four threats.

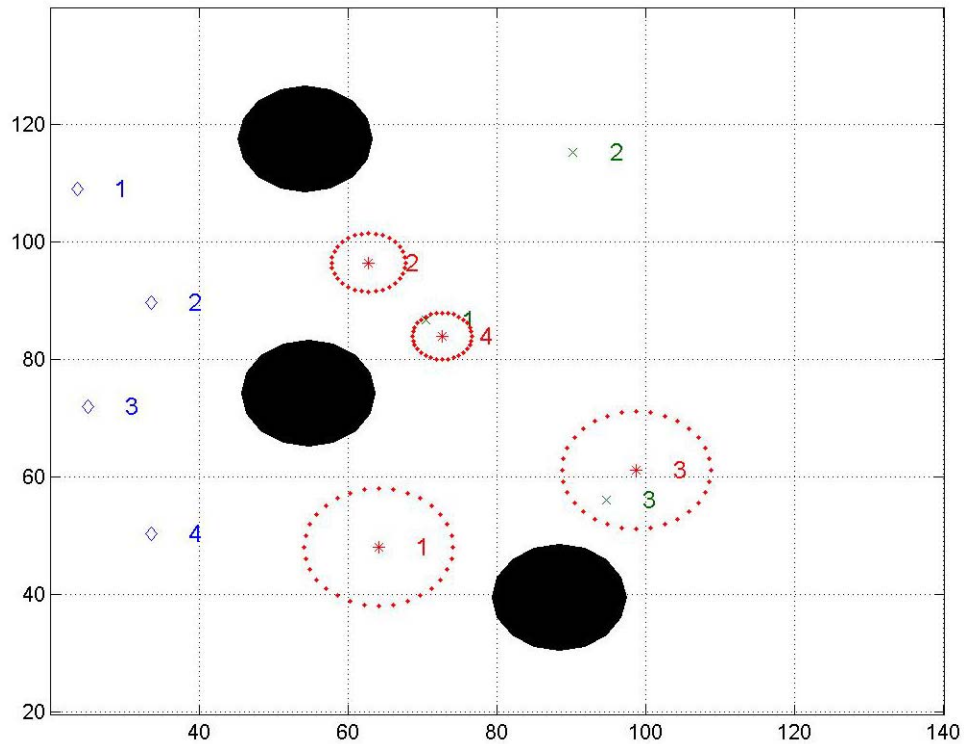

Figure 6.11: Initial battlefield setup for decentralized simulation example

Once the user sets up the initial battlefield, the simulation proceeds just as the centralized version would. As events occur, the MATLAB command window prints them, and replans occur. Once the simulation completes, the user can choose the PLOT SIMULATION button shown at the top left of the main simulation system in Figure 6.1. This produces the moving plot being discussed. This plot shows the UAVs in motion traveling toward their assigned targets, and shows dynamic environment changes such as pop-up targets, pop-up threats, removal of destroyed targets, and loss of UAVs. An option is also given with this plotting to show the path the individual UAVs have traveled thus far. These traveled paths reveal information about where the UAVs were located at times of replan and which targets they have been assigned to. Figure 6.12 illustrates a captured frame of this moving plot.

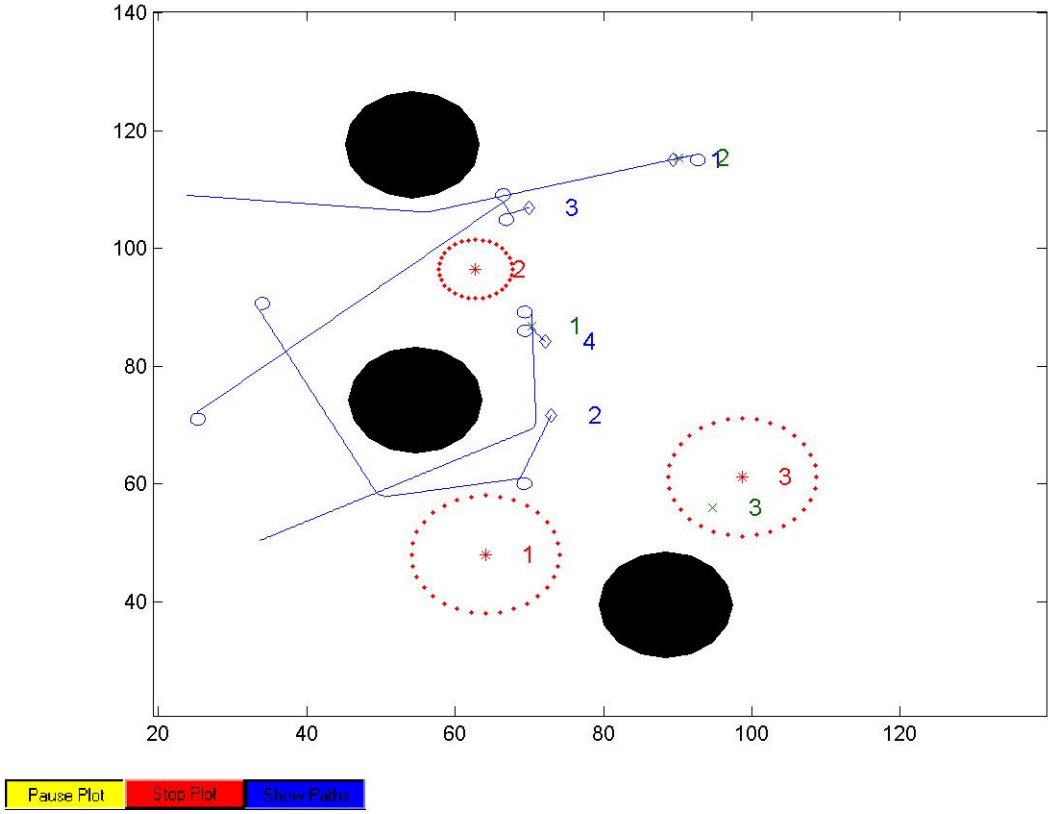

Figure 6.12: Decentralized simulation example

In this specific frame, two replans have already occurred. UAV 1 has confirmed target 2 is a target and is currently assessing the target; UAV 2 was first assigned to target 3 but has now been reassigned to target a; UAV 3 was initially assigned to target 2 along with UAV 1, was later assigned to target 3, and finally has been reassigned back to target 2; and lastly, UAV 4 has completed assessing target 1 and is now assigned to target 3. It can also be seen that threat 4, which was an antiaircraft artillery piece guarding target 1, has fired unsuccessfully at the only UAV to have entered its effective range – UAV 4.

The decentralized simulation environment proposed by research objective 5 has now been completed. The next and last discussion chapter will be dedicated to comparison of the centralized and decentralized simulations in terms of 'real-time' simulation; furthermore, communication issues will be addressed for this decentralized simulation environment.

Chapter 7
Comparison of Decentralized and Centralized Simulations

7.1 Simulation Efficiency

Real-time performance is crucial for implementation of any scheme aboard an aircraft. This section investigates all MATLAB codes in terms of time of completion, and both SIMULINK simulations are run in conjunction with a simulation profiler that shows how much time is spent executing the simulation.

The MATLAB code that performs the path planning and task allocation approach discussed in Chapter 3 can be used with MATLAB function *profile* to track program execution time. The results of running the *path_planning.m* code with the MATLAB Profiler is shown in the next four tables for three different cases. The first of these tables gives a summary of the profile reports, such as number of UAVs, targets, threats, and no-fly zones used to generate the profile report, in which table the report is found in, and the total recorded time the *path_planning.m* code took to execute. The next three tables present the profile report generated for each of the three cases.

Table 7.1: Summary of MATLAB Profile Reports

Number of UAVs	4	5	9
Number of Targets	4	5	9
Number of Threats	4	5	15
Number of No-fly Zones	4	5	15
Profile Report found in:	Table 4	Table 5	Table 6
Total recorded time:	1.41 s	3.10 s	20.48 s
Number of M-functions:	30	30	30
Number of M-subfunctions:	2	2	2
Number of MEX-functions:	1	1	1
Clock precision:	0.00000006 s	0.00000006 s	0.00000006 s
Clock Speed:	1584 Mhz	1584 Mhz	1584 Mhz

Table 7.2: Profile Report based on 4 UAVs, 4 Targets, 4 Threats, and 4 No-fly Zones

Name	Time		Calls	Time/call
path_shrtng	1.11100000	78.7%	1	1.11100000000
shorten_paths	0.88100000	62.4%	16	0.05506250000
cheapest_paths	0.16100000	11.4%	1	0.16100000000
vrn_diag_gen	0.11000000	7.8%	1	0.11000000000
update_cost	0.09000000	6.4%	16	0.00562500000
dijk	0.08100000	5.7%	16	0.00506250000
heading_angle_paths	0.06000000	4.2%	16	0.00375000000
voronoi	0.06000000	4.2%	1	0.06000000000
delaunay	0.04000000	2.8%	1	0.04000000000
delaunayn	0.03000000	2.1%	1	0.03000000000
pred2path	0.02100000	1.5%	16	0.00131250000
vrt_sim_convert	0.02000000	1.4%	1	0.02000000000
list2adj	0.02000000	1.4%	1	0.02000000000
c_assign	0.02000000	1.4%	1	0.02000000000
set_thc	0.02000000	1.4%	1	0.02000000000
unique	0.02000000	1.4%	2	0.01000000000
perms	0.01000000	0.7%	4	0.00250000000
mmkp_new	0.01000000	0.7%	1	0.01000000000
mmkp_task_allocation	0.01000000	0.7%	1	0.01000000000
cart2pol	0.01000000	0.7%	136	0.00007352941
fillet_path	0.01000000	0.7%	16	0.00062500000
connect_vrn	0.01000000	0.7%	2	0.00500000000
voronoi/circle	0.01000000	0.7%	2	0.00500000000
sortrows	0.01000000	0.7%	2	0.00500000000
profile	0.00000000	0.0%	1	0.00000000000
pol2cart	0.00000000	0.0%	136	0.00000000000
isint	0.00000000	0.0%	2	0.00000000000
num2cell	0.00000000	0.0%	1	0.00000000000
mat2vec	0.00000000	0.0%	1	0.00000000000
qhullmx	0.00000000	0.0%	1	0.00000000000
sortrows/sort_back_to_front	0.00000000	0.0%	2	0.00000000000
nargchk	0.00000000	0.0%	39	0.00000000000
filter_zeros	0.00000000	0.0%	4	0.00000000000

Table 7.3: Profile Report based on 5 UAVs, 5 Targets, 5 Threats, and 5 No-fly Zones

Name	Time		Calls	Time/call
path_shrtng	2.72400000	87.8%	1	2.72400000000
shorten_paths	2.35300000	75.8%	25	0.09412000000
cheapest_paths	0.22000000	7.1%	1	0.22000000000
update_cost	0.16000000	5.2%	25	0.00640000000
dijk	0.14000000	4.5%	25	0.00560000000
heading_angle_paths	0.12000000	3.9%	25	0.00480000000
vrn_diag_gen	0.11000000	3.5%	1	0.11000000000
voronoi	0.06000000	1.9%	1	0.06000000000
delaunay	0.05000000	1.6%	1	0.05000000000
delaunayn	0.04000000	1.3%	1	0.04000000000
vrt_sim_convert	0.03000000	1.0%	1	0.03000000000
unique	0.03000000	1.0%	2	0.01500000000
mmkp_new	0.02000000	0.6%	1	0.02000000000
mmkp_task_allocation	0.02000000	0.6%	1	0.02000000000
pol2cart	0.02000000	0.6%	481	0.00004158004
fillet_path	0.02000000	0.6%	25	0.00080000000
list2adj	0.02000000	0.6%	1	0.02000000000
c_assign	0.02000000	0.6%	1	0.02000000000
set_thc	0.02000000	0.6%	1	0.02000000000
cart2pol	0.01000000	0.3%	481	0.00002079002
voronoi/circle	0.01000000	0.3%	2	0.00500000000
isint	0.01000000	0.3%	2	0.00500000000
num2cell	0.01000000	0.3%	1	0.01000000000
mat2vec	0.01000000	0.3%	1	0.01000000000
sortrows	0.01000000	0.3%	2	0.00500000000
profile	0.00000000	0.0%	1	0.00000000000
perms	0.00000000	0.0%	5	0.00000000000
pred2path	0.00000000	0.0%	25	0.00000000000
connect_vrn	0.00000000	0.0%	2	0.00000000000
qhullmx	0.00000000	0.0%	1	0.00000000000
sortrows/sort_back_to_front	0.00000000	0.0%	2	0.00000000000
nargchk	0.00000000	0.0%	57	0.00000000000
filter_zeros	0.00000000	0.0%	4	0.00000000000

Table 7.4: Profile Report based on 9 UAVs, 9 Targets, 15 Threats, and 15 No-fly Zones

Name	Time		Calls	Time/call
path_shrtng	15.46200000	75.5%	1	15.462000000000
shorten_paths	13.88000000	67.8%	81	0.171358024691
mmkp_task_allocation	4.03600000	19.7%	1	4.036000000000
mmkp_new	4.02600000	19.7%	1	4.026000000000
perms	1.02200000	5.0%	9	0.113555555556
cheapest_paths	0.82100000	4.0%	1	0.821000000000
update_cost	0.71100000	3.5%	81	0.008777777778
dijk	0.71100000	3.5%	81	0.008777777778
heading_angle_paths	0.43100000	2.1%	81	0.005320987654
vrn_diag_gen	0.12000000	0.6%	1	0.120000000000
cart2pol	0.07000000	0.3%	2801	0.000024991075
voronoi	0.06000000	0.3%	1	0.060000000000
delaunay	0.05000000	0.2%	1	0.050000000000
pol2cart	0.05000000	0.2%	2801	0.000017850768
pred2path	0.05000000	0.2%	81	0.000617283951
fillet_path	0.04000000	0.2%	81	0.000493827160
delaunayn	0.04000000	0.2%	1	0.040000000000
vrt_sim_convert	0.04000000	0.2%	1	0.040000000000
c_assign	0.04000000	0.2%	1	0.040000000000
set_thc	0.04000000	0.2%	1	0.040000000000
unique	0.03000000	0.1%	2	0.015000000000
num2cell	0.01000000	0.0%	1	0.010000000000
mat2vec	0.01000000	0.0%	1	0.010000000000
list2adj	0.01000000	0.0%	1	0.010000000000
connect_vrn	0.01000000	0.0%	2	0.005000000000
voronoi/circle	0.01000000	0.0%	2	0.005000000000
profile	0.00000000	0.0%	1	0.000000000000
isint	0.00000000	0.0%	2	0.000000000000
qhullmx	0.00000000	0.0%	1	0.000000000000
sortrows/sort_back_to_front	0.00000000	0.0%	2	0.000000000000
sortrows	0.00000000	0.0%	2	0.000000000000
nargchk	0.00000000	0.0%	169	0.000000000000
filter_zeros	0.00000000	0.0%	4	0.000000000000

As shown in Table 7.1, a case where there are only four UAVs executes quickly in 1.41 seconds. This time represents the necessary time for the code to complete once started. This time is of course a function of processor speed and memory. All figures shown here were performed with a 1.6 GHz processor and 256 MB of RAM. However, completion time is not just a function of computer hardware, but also the initial problem set up. Whenever the problem is extended to 5 UAVs, 5 targets, 5 threats, and 5 no-fly zones, the program takes 3.10 seconds to complete. Whenever the problem is extended to the maximum allowable inputs of 9 UAVs, 9 targets, 15 threats, and 15 no-fly zones, the simulation takes over 20 seconds to output all paths and assignments!

The reason behind the greatly increased computing time can be seen by the number of permutations experienced by increasing the number of UAVs. With a standard simulation of 4 UAVs performing 4 assignments, there are only 16 different combinations of UAV to assignment. For 5 UAVs, that number increases to 120. For 6 UAVs there are 720 permutations, 7 UAVs have 5040 permutations, and for 8 UAVs there are 40,520 permutations. Whenever 9 different UAVs are used in a single team and each must have a different assignment, there are 362,880 possible combinations of UAV to assignment! For the simulation with 4 UAVs, the MMKP section takes 0.7% of the total completion time to execute. For the 5 UAV simulation, MMKP takes roughly the same percentage of time, decreasing slightly to 0.6%. However, for the 9 UAV simulation, MMKP takes 19.7% of the completion time to determine the optimal combination of UAVs to assignments. For this reason, the limit of the UAVs and targets in simulation was chosen to be 9 each. Since the complexity of permutations is a factorial function, a path planning and task allocation scheme for 10 UAVs would have 3,628,880 permutations, 11 UAVs would have 39,916,800 permutations, and 12 UAVs would encounter 479,001,600 different combinations of UAV to assignment.

A second reason for the increased computation time for higher UAV systems is the number of paths that have to be shortened and made flyable. For the 4 UAV simulation, there are only 16 paths, for 5 UAVs there is 25 paths, and for 9 UAVs, there are 81 paths. The time required to shorten and make flyable the paths also depends on

how complex the system is. If there are a high number of UAVs but a low number of threats and no-fly zones, the paths can quickly be optimized. For a high number of obstacles to fly around, this time increases. Path shortening can be seen in Tables 7.2-7.4 to take roughly 70% of the total completion time, indicating an approximate linear function to complexity associated with path shortening.

For standard simulations with a limited number of UAVs and targets (such as 4 or 5), the path planning and task allocation MATLAB code computes in only a few seconds, indicating that it could be used in real aircraft systems. MATLAB code is also a slower computational environment and turning this code into an executable C code will speed up completion time even further. In situations with near maximum numbers of UAVs, targets, threats, and no-fly zones are desired, there are two possible options for quicker completion time of task assignments. First, the team of UAVs could be broken into two smaller teams that cooperate to perform tasks, so essentially there would be two teams of 4 or 5 with each team performing 4 or 5 assignments. Secondly, the path optimization (shortening and flyability) can be performed after the assignments are chosen. This would cause the completion time of the code to be reduced by about 50%. Performing path optimization before allocating tasks is beneficial to choosing an optimal assignment. For a standard number of UAVs, targets, threats, and no-fly zones, the degraded performance is not worth the trade off for a shorter computational time where paths are shortened and made flyable post-assignment. In large simulations, giving up some optimality for much faster running time should be considered.

Execution times for simulation is also of interest. SIMULINK has a simulation profiler built into its Performance Tools option. This simulation profile generates a profile report similar to the MATLAB profile report, detailing the execution time of a simulation. The decentralized and centralized simulations were both run with this tool, and the findings are presented next. To ensure equitable conditions when comparing these two simulations, the same initial battlefield was used for both. This battlefield is show in the following figure and uses 4 UAVs, 3 targets, 3 no-fly zones, and 4 threats.

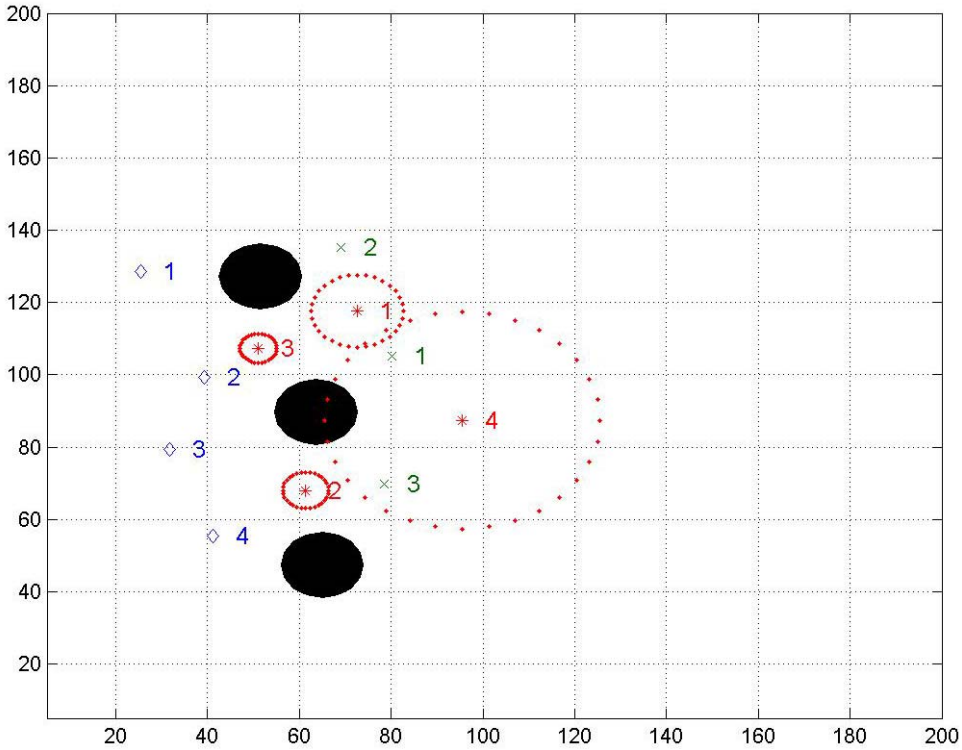

Figure 7.1: Initial battlefield setup for SIMULINK Profile Reports

The centralized simulation was first executed using the profile function. The simulation was tested for running the initialization of the simulation and the first 10 simulated second. Table 7.5 shows the results of running this simulation normally within SIMULINK, and also with the Accelerator function.

Table 7.5: SIMULINK Profile Summary for centralized simulation

Simulation Speed	Normal	Accelerator
Total recorded time:	18.03 s	4.90 s
Number of Block Methods:	1471	76
Number of Internal Methods:	9	5
Number of Nonvirtual Subsystem Methods:	104	4
Clock precision:	0.00000006 s	0.00000006 s
Clock Speed:	1584 Mhz	1600 Mhz

The SIMULINK Accelerator produces an executable C file that replaces the simulation used within SIMULINK. The completion time of the simulation to initialize and run for 10 simulated seconds was 4.90 seconds with the Accelerator function, and 18.03 seconds when the simulation was executed as normal. Tables 7.6 and 7.7 detail the profile report for the normal execution and the Accelerator execution, respectively. For the normal execution, the initialization of the simulation task 35% of the completion time, or 6.3 seconds. The rest of the time is used for executing the simulation for 10 simulated seconds, which occurred in 11.7 seconds.

Table 7.6: SIMULINK Profile Report for centralized version

Name	Time		Calls	Time/call
sim	18.02600000	100.0%	1	18.02600000000
ModelExecute	11.66600000	64.7%	1	11.66600000000
pathplan (Output)	8.46200000	46.9%	205	0.04127804878
MajorOutputs	8.46200000	46.9%	205	0.04127804878
ModelInitialize	6.30900000	35.0%	1	6.30900000000
Integrate	2.46200000	13.7%	202	0.01218811881
pathplan (MinorOutput)	2.14100000	11.9%	210	0.01019523810
MinorOutputs	2.14100000	11.9%	210	0.01019523810

The Accelerator-based simulation ran in 4.9 seconds. The model initialization took over half of the completion time, representing 2.7 seconds. The simulation ran for 10 simulated seconds afterward in 2.2 seconds.

Table 7.7: SIMULINK Profile Report for centralized version, with Accelerator

Name	Time		Calls	Time/call
sim	4.89700000	100.0%	1	4.89700000000
ModelInitialize	2.71400000	55.4%	1	2.71400000000
ModelExecute	2.14300000	43.8%	1	2.14300000000
pathplan (Output)	1.81300000	37.0%	205	0.00884390244

The same steps were used with the decentralized simulation. As shown in Figure 7.1, the same battlefield setup was used for both simulations. As with the centralized version, a normal simulation and a SIMULINK Accelerator-based simulation were initialized and ran for 10 simulated seconds. Table 7.8 shows both summaries for the two simulations of the decentralized version.

Table 7.8: SIMULINK Profile Summary for decentralized simulation

Simulation Speed	Normal	Accelerator
Total recorded time:	63.05 s	37.37 s
Number of Block Methods:	2965	160
Number of Internal Methods:	9	5
Number of Nonvirtual Subsystem Methods:	455	4
Clock precision:	0.00000006 s	0.00000006 s
Clock Speed:	1600 Mhz	1600 Mhz

The decentralized simulations took considerably longer to execute than their centralized counterparts. For the normal simulation, initialization and 10 simulated seconds took 63 seconds to complete. For the Accelerator-based version, this took 37 seconds. Tables 7.9 and 7.10 detail the two profile reports.

Table 7.9: SIMULINK Profile Report for decentralized version

Name	Time		Calls	Time/call
`sim`	63.05100000	100.0%	1	63.05100000000
`ModelExecute`	53.03700000	84.1%	1	53.03700000000
`pathplan (Output)`	46.30200000	73.4%	201	0.23035820896
`MajorOutputs`	46.30200000	73.4%	201	0.23035820896
`ModelInitialize`	9.92400000	15.7%	1	9.92400000000
`Integrate`	3.29100000	5.2%	200	0.01645500000
`pathplan (MinorOutput)`	2.95100000	4.7%	200	0.01475500000
`MinorOutputs`	2.95100000	4.7%	200	0.01475500000

Table 7.10: SIMULINK Profile Report for decentralized version, with Accelerator

Name	Time		Calls	Time/call
sim	37.37300000	100.0%	1	37.37300000000
ModelExecute	24.26500000	64.9%	1	24.26500000000
pathplan (Output)	22.00300000	58.9%	201	0.10946766169
ModelInitialize	13.01800000	34.8%	1	13.01800000000

For the normal simulation, the initialization took 10 seconds and the Accelerator-based simulation initialized in 13 seconds. The increase in initialization times represents the increased from a single centralized simulation to 9 independent UAV simulations. Therefore, this increase in initialization is expected. The execution times were then 53 seconds and 24.3 seconds, respectively. It should here be noted that the profile function itself is quite computationally expensive to simulate. About 15 seconds at the beginning of the simulation can be attributed to the initial path planning. Because the simulation is setup for a maximum of nine UAVs, each of these possible UAVs run a path-planning scheme even if they do not exist. This accounts for the first 15 seconds after the initialization. However, without the profiler running, 10 simulated seconds was found to run in 6.62 seconds for the normal simulation. The profile shows 38 seconds for this part for the normal simulation, and 9 seconds for the Accelerator-based simulation.

Though the decentralized simulation has been shown to take longer to simulate a given system, an interesting aspect is found when considering that the decentralized simulation consists of essentially 9 UAVs being simulated by the same central processor (a personal computer). Since the objective is to achieve real-time performance for an individual UAV simulation, the individual UAV system needs to be investigated, not the entire team being run by a central processor. Since a single CPU cannot run simulations in parallel, the time for an individual UAV system is approximately one-ninth of the total simulation time for the decentralized simulation. This computes to seven seconds for the normal simulation and just over four seconds for the Accelerator-based simulation. These times are even faster than the centralized version, and with reason. Since the individual UAVs within the decentralized simulation do not have to perform calculation

regarding the other UAVs (with respect to dynamics and threats and no-fly zone checking), the simulation should occur in less time.

7.2 Miscommunication

Just like the real-time performance of software, investigation of real-life situations using simulation is crucial. For decentralized path planning and task allocation, the critical link for correct decision making is communication amongst a team of UAVs. The next three sections investigate three possible scenarios where problems in communications can lead to incorrect decisions for the team of UAVs.

The first possible problem with communication is miscommunication. There are two possible ways for miscommunication to occur. The first way would be a fault within the aircraft's software or hardware to either send out incorrect signals or misinterpret signals from other aircraft. This is less likely to occur than the second way, which is caused by enemy electronic warfare efforts. If this electronic warfare leads to some uncertainty, say within the exact locations of other team members, then the individual UAVs may base their path planning on wrong information.

Miscommunication leads to incorrect decision on the part of the individual UAVs with a cooperating team. The likely outcome of miscommunication is that certain tasks will be duplicated by multiple UAVs while other tasks will be neglected. To test the effects of miscommunication, the decentralized simulation was modified as shown in Figure 7.2. A noise generator was added to the communications about UAV positions, so that individual UAVs would not know the location of their team members within a few kilometers. Small allowances within aircraft position will not cause any incorrect decisions, but the difference of several kilometers can.

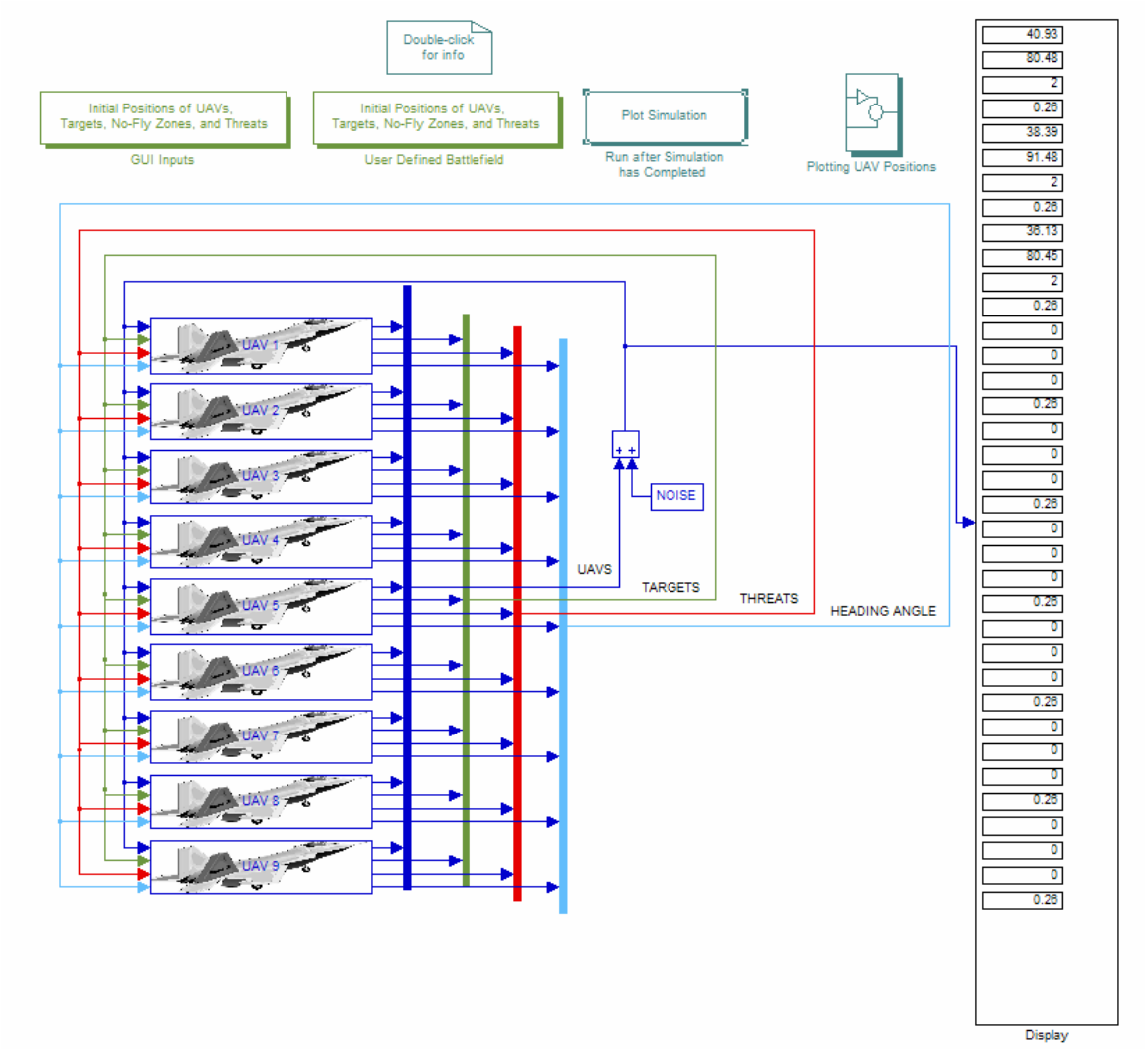

Figure 7.2: Main system for decentralized UAV control with miscommunication

The noise that is added to the positions of each UAV follows a Gaussian probability density function with a mean of zero and a standard deviation of 1. This noise is run through a gain of value 2, so each UAV's position can be plus or minus 2 kilometers in the X-direction and plus or minus 2 kilometers in the Y-direction. Figure 7.3 illustrates the NOISE block of the main system, and Figure 7.4 shows the noise generators and gains for each individual UAV of the team.

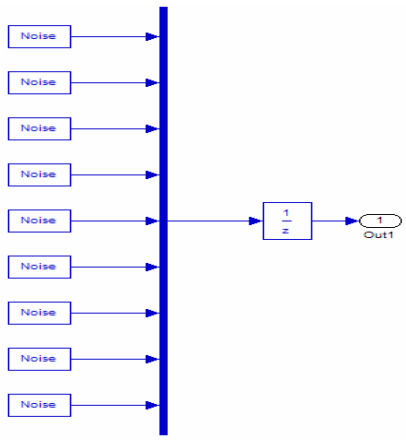

Figure 7.3: NOISE block used for simulating miscommunication

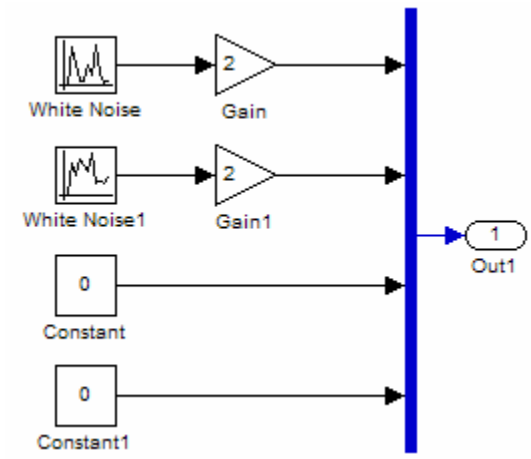

Figure 7.4: Individual UAV noise

These modifications were used to test the response to misinformation. An example is presented here for the simple simulation of 3 UAVs, 2 targets, 2 no-fly zones, and a single threat. Figure 7.5 contains the initial battlefield setup for this example.

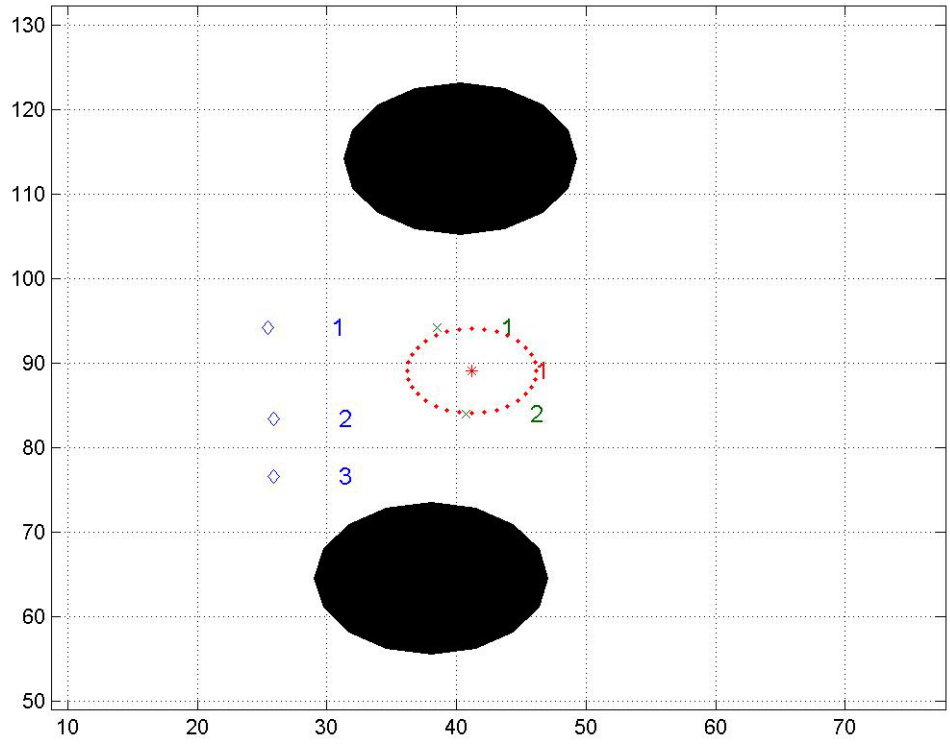

Figure 7.5: Initial battlefield setup for miscommunication example

The UAVs are initialized with the correct information, so the simulation proceeds correctly until the first replan occurs. Whenever this replan occurs, UAV 2 is assessing target 1 while UAVs 1 and 3 are assigned to target 2. The replan contains incorrect information for the locations of all three UAVs. This incorrect information causes all three UAVs to be assigned to target 2, while no UAV is assigned to target 1. Tasks are still being accomplished, but the simulation will take longer overall because certain tasks are being neglected. Figure 7.6 shows the UAVs after the replanning. One should note the aerodynamic path discontinuities for the UAVs. The moving plot shown here is based upon the UAVs knowledge of positions, and whenever noise causes the positions to be distorted during a replan, the paths become strange and certainly dynamically unfeasible. However, the dynamics of the aircraft do not see these discontinuities, since they only represent noise that makes the plot somewhat incorrect.

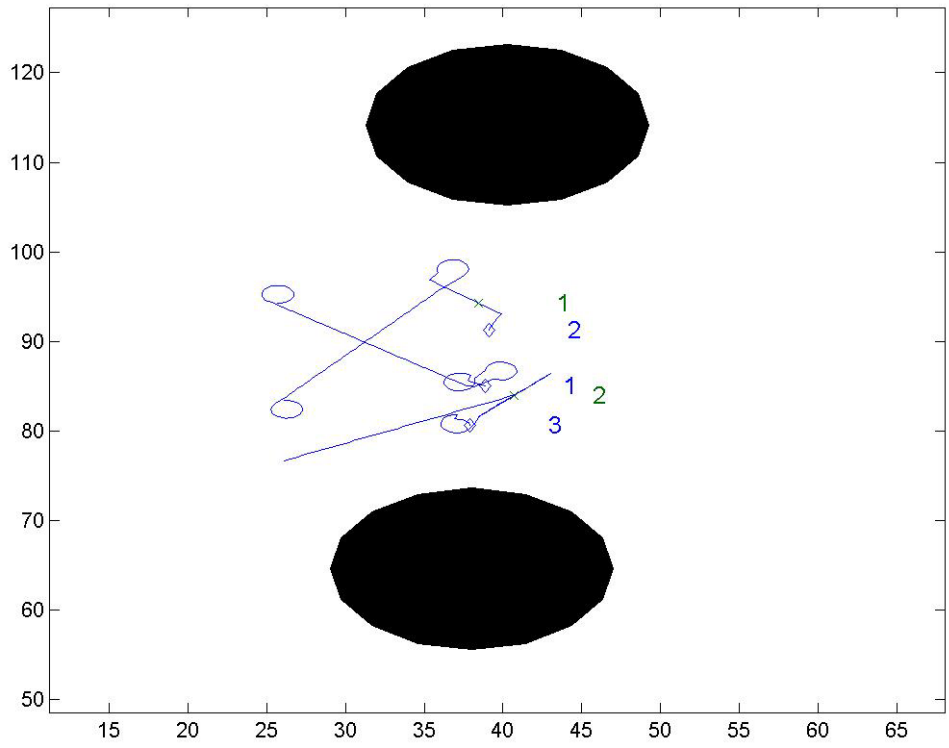

Figure 7.6: Miscommunication, decentralized simulation example

7.3 Delay of Communication

Delay of communication is the second type of investigated problems with communication. Delays are already inherent within the situation, as delays can be quite useful for initialization purposes and comparison of old information with current information. However, longer delays within the communications will certainly cause incorrect decisions. Longer delays can be seen as essentially a loss of communication that occurs for a definite period of time. Loss of communication will be investigated in the next section, and an example will be presented as well. Delays in communication will respond in the exact same manner, with multiple UAV assignments of a single task while the team neglects other tasks.

7.4 Loss of Communication

The third source for problems in communication is loss of communication. Loss of communication would typically result from highly effective enemy electronic warfare, which would produce an environment where all communications are effectively jammed. Loss of communication could also result from damage to an individual UAV, but not enough damage to cause destruction of the UAV or inability to perform tasks.

In any situation, one or more UAVs can experience loss of communication. The UAVs that loose communication effectively become a separate, one vehicle team from the other group. The lone UAVs will still see teammates where their last known position was, and it will still be assumed they will perform tasks, but when no communication about task accomplishment is received, the lone UAV performs all known tasks on all the known targets. Meanwhile, for the team of UAVs that has lost contact with one or more members, these members will essentially be seen as UAVs whose last known coordinates represent their location. These lost UAVs will still be expected to perform tasks as before, but because no information is received from them, their tasks are eventually delegated to other team members who still properly communicate with the team. From these two scenarios, the omniscient user sees a group of UAVs performing tasks, and one or more lone UAVs who are attempting to duplicate those same tasks, whether they have been performed or not. Thus, typically there are multiple UAVs performing the same task while other tasks are neglected, as has been seen in the miscommunication case.

An example can be shown representing this scenario. The decentralized simulation must first be modified to account for a loss of communication. Figure 7.7 shows the modification to the decentralized scheme where UAV 2 has lost communication with the group.

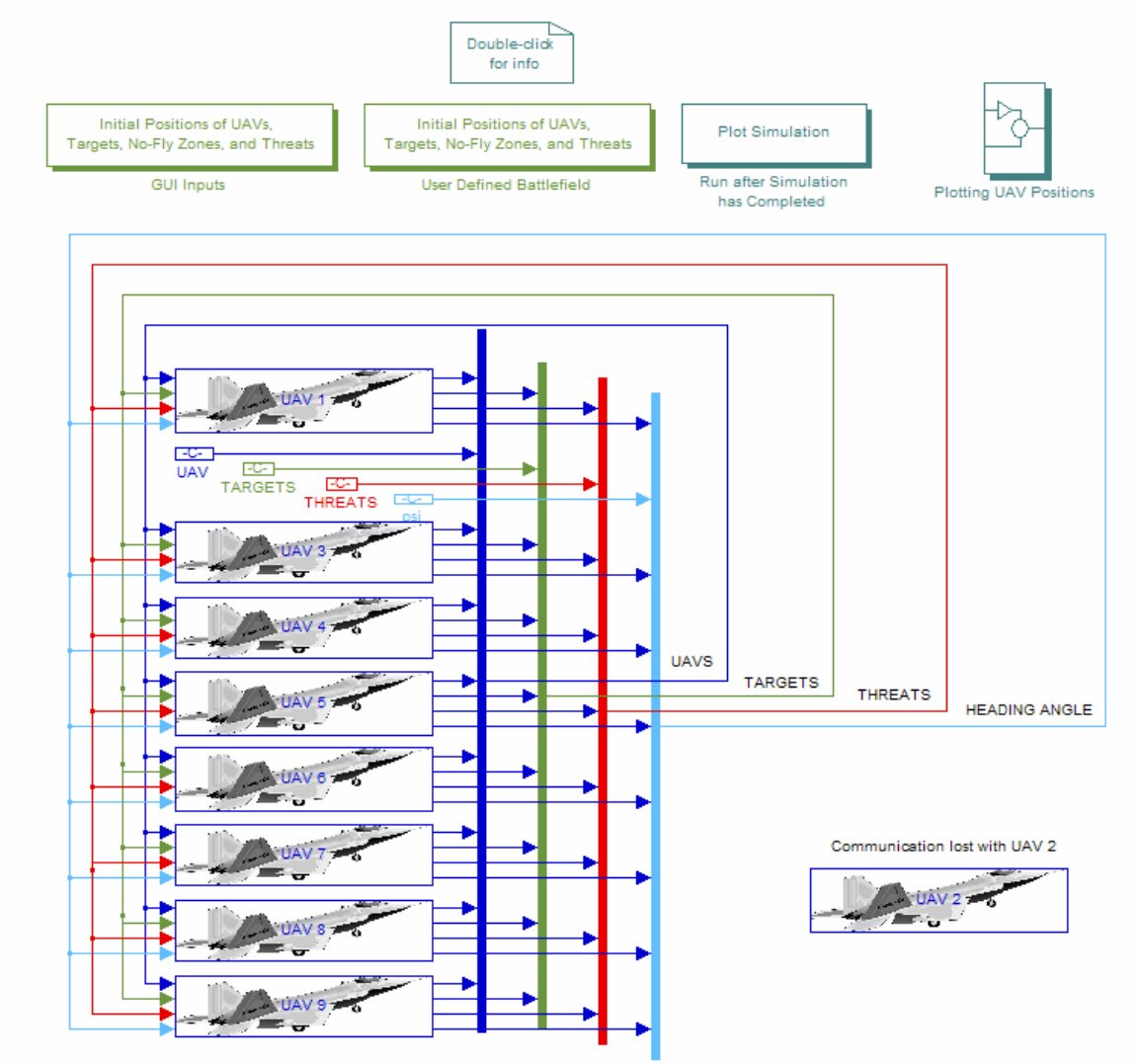

Figure 7.7: Main system with individual UAV communication loss

The group of UAVs remains the same, but in place of UAV 2 are now just the original coordinates of the vehicle. The group sees this UAV as one who continuously remains at its initial position, but not as one who has been destroyed (because the loss of communication may just be temporary). UAV 2 is now acting like a team by itself. Though it sees the rest of the group as not being destroyed, the group essentially stays at their original coordinates. Figure 7.8 shows the modifications for the individual system to allow for simulation of this isolation.

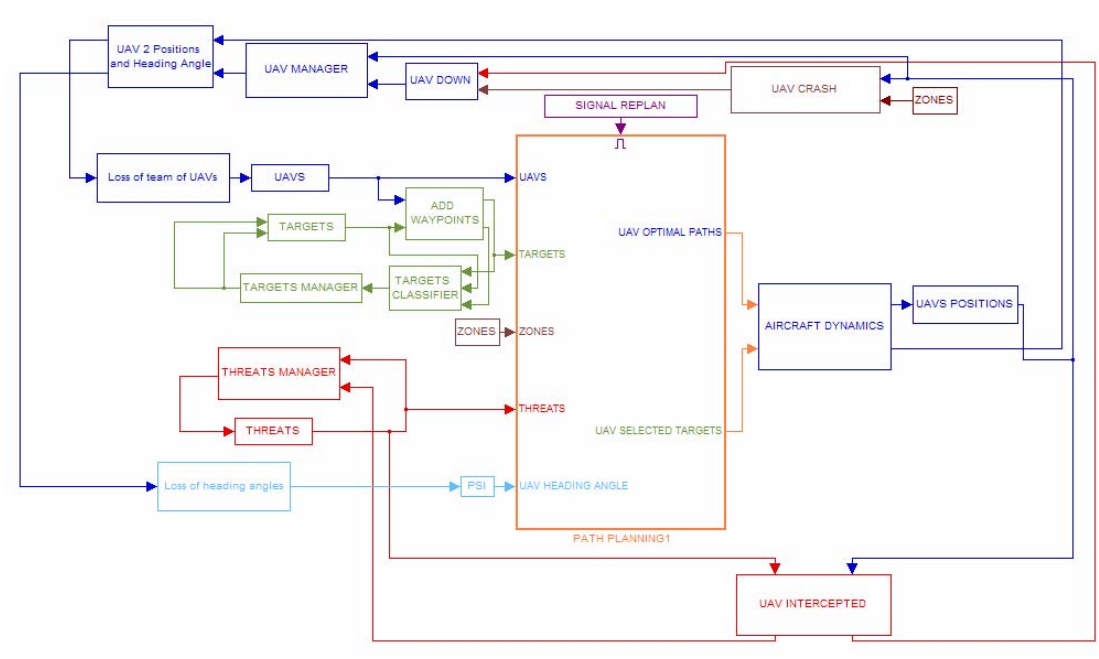

Figure 7.8: Main system for individual UAV 2, showing modifications

The UAV uses its own known coordinates and target and threat states, and uses two new systems to simulate this loss of a team. These two blocks contain the system shown in Figure 7.9. These systems show the UAV team as stationary at their original coordinates. The team members are still expected by UAV 2 to perform tasks, but because UAV 2 sees them as never accomplishing those tasks, eventually UAV 2 will perform all the known target assignments.

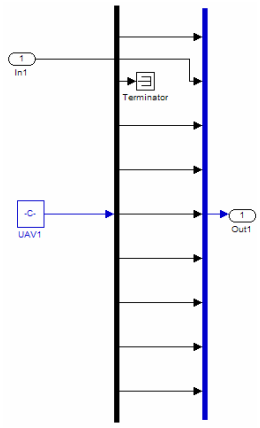

Figure 7.9: Loss of team of UAVs block

An example is now shown using this new simulation. The initial battlefield is given by the following figure. There are 3 UAVs, 2 targets, 2 no-fly zones, and a single threat.

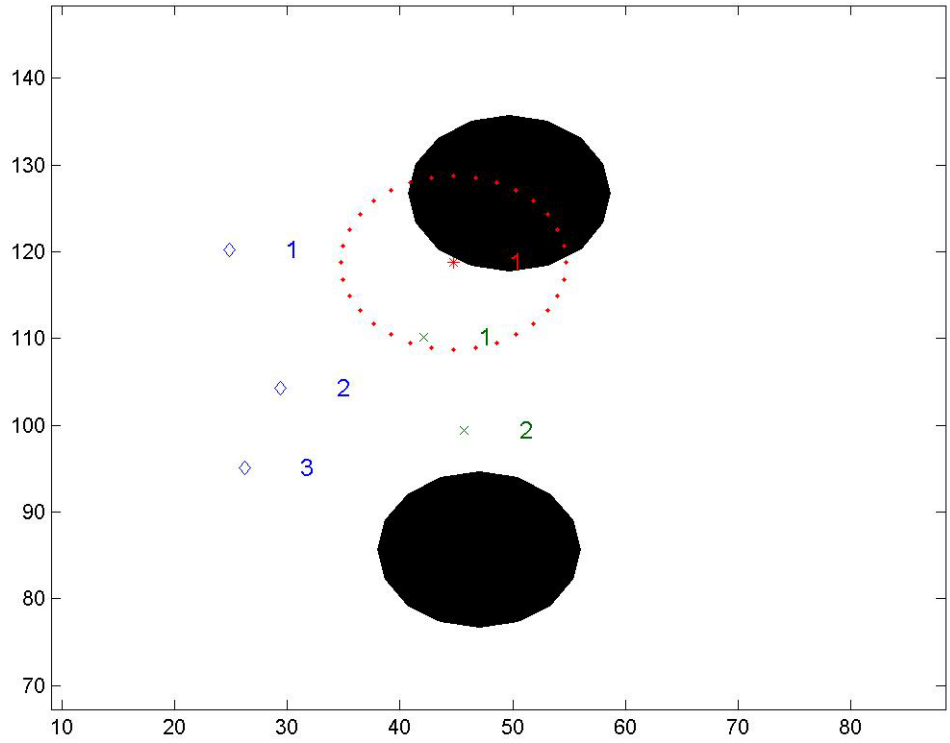

Figure 7.10: Initial battlefield setup for individual communication loss example

The simulation begins with all UAVs knowing the correct initial positions. The UAVs make the correct decisions of UAVs 1 and 2 being assigned to the higher-valued target, target 2, while UAV 3 is assigned to target 1. However, target 1 lies inside of a Crotale SAM's effective range, and whenever UAV 3 crosses that boundary, it is destroyed. At this point, UAV 1 believes UAV 2 still exists at its original position, which is the last known position for UAV 2. Whenever UAV 1's path planning and task allocation scheme runs, UAV 1 is again assigned to target 2, while UAV 2 is expected to perform target reconnaissance on target 1. Meanwhile, UAV 2 has lost communication with the other two UAVs. Therefore, UAV 2 simply continues on for its assigned task at target 2, because UAV 2 never receives communication that UAV 3 gets destroyed. The end result is shown in Figure 7.11.

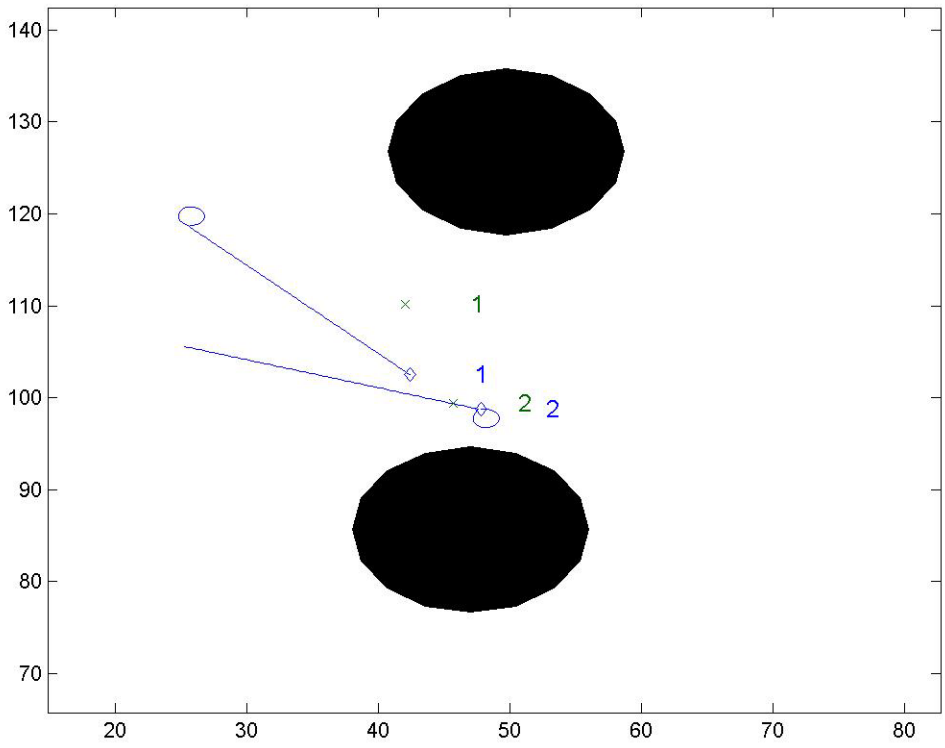

Figure 7.11: Individual communication loss example

As seen here, both UAV 1 and UAV 2 are assigned to target 2. Neither UAV has assigned tasks at target 1, because of the lack of communication. UAV 1 expects UAV 2 to perform tasks on target 1, while UAV 2 expects the now destroyed UAV 3 to perform tasks on target 1. The result of this loss of communication is a lack of cooperation. Tasks are still performed, even if duplicated, and eventually all tasks will be completed (assuming there is at one surviving UAV to perform assignments). The decentralized scheme allows the UAVs to make their own decisions, even if incorrect because of problems with communication. Even with incorrect decision making on the individual UAV parts, missions can still be accomplished, whereas with a centralized scheme, all UAVs would be lost once proper communication ceased.

Chapter 8
Conclusions and Recommendations

8.1 Conclusions

The research effort presented here accomplished the six research objective as stated at the end of the Introduction chapter. The first objectives were to create a path planning and task allocation scheme. This scheme began by using Voronoi diagram to connect UAVs to targets with graphical edges. These edges next had costs assigned to them based on their length and possible threat cost. Once edge costs were assigned, Dijkstra's algorithm was used to search the graph edges to determine the lowest-cost path for each permutation of UAV to target. These lowest-cost paths were then further refined by shortening using a line of sight method, adding fillets along the edge intersections, and adding initial sections to the path to transition the current UAV heading angle to the desired one. The last step in the path planning and task allocation scheme was to use a Multi-dimensional, Multiple-Choice Knapsack Problem solution to allocate all assignments while minimizing UAV team costs.

The next research addressed the third and fourth objects by development of a SIMULINK-based centralized simulation environment. This simulation used the path planning and task allocation scheme previously developed, and added time-varying, dynamic environment, aspects. Pop-up target and threat capabilities were implemented. A UAV manager was developed to address the possibilities of individual or multiple UAV loss. A UAV model was implemented with an aircraft dynamics subsystem. Target states were used to track the tasks performed on individual targets, and real-possibilities were modeled to include objects disguised as targets, and targets that are not destroyed in the first attack.

The latter part of the research effort was focused on development of a decentralized simulation environment to complete the last research objectives. This decentralized version now includes a vehicle's own decision making capabilities and

communication amongst vehicles. Next, the decentralized simulation was compared with the centralized version in terms of simulation efficiency. It was concluded that the path planning and task allocation scheme could be implemented in a real-time environment only for a limited number of UAVs, targets, threats, and no-fly zones, as expected. The centralized simulation proved to be a faster simulation than the decentralized version, but when the decentralized is considered to be essentially running nine separate simulations at once, the individual UAV simulations show faster times than the centralized version. Lastly, real communications issues were addressed to show that while communication problems lead to a lack of cooperation, tasks can still be performed and missions completed within the decentralized simulation environment.

8.2 Recommendations

From this research effort, further investigation and implementation of this decentralized path planning and task allocation scheme could be pursued in several directions. The first direction would be conversion of the decentralized simulation environment into executable files in C code. These executable files could be tested using parallel processing to truly model a team of UAVs cooperating. The next direction this research could be taken in would include small, inexpensive UAVs. These UAVs could be used as a proving vehicle for this approach, to show the actual implementation of this decentralized path planning and task allocation scheme.

References

1. UAV Forum. "Unmanned Aerial Vehicles and Precision Guided Munitions at the Centennial" http://www.uavforum.com/library/defnews.doc

2. Department of Defense. *Unmanned Aerial Vehicles Roadmap 2002-2027*. Office of the Secretary of Defense: December 2002.

3. UAV Forum. "Librarian's Desk – UAV forum" http://www.uavforum.com/library/librarian.htm

4. McLain, T.W., "Coordinated Control of Unmanned Air Vehicles" Air Vehicles Directorate, Wright-Patterson Air Force Base, Ohio, summer 1999.

5. USAF ARFL - Air Vehicles Directorate. "Unmanned Air Vehicles" http://www.va.afrl.af.mil/FA/UAV/uav_index.html

6. FAS Military Analysis Network. "Low Cost Autonomous Attack System (LOCAAS) Miniature Munition Capability" http://www.fas.org/man/dod-101/sys/smart/locaas.htm

7. Phantom Works. "Unmanned Combat Air Vehicle (X-45)" http://www.boeing.com/phantom/ucav.html

8. Bortoff, S.A., "Path-Planning for Unmanned Air Vehicles" Air Vehicles Directorate, Wright-Patterson Air Force Base, Ohio, August 1999.

9. McLain, T.W., "Cooperative Control of UAV Rendezvous" Air Vehicles Directorate, Wright-Patterson Air Force Base, Ohio, Summer 2000.

10. McLain, T.W., and Beard, R.W., "Trajectory Planning for Coordinated Rendezvous of Unmanned Air Vehicles" AIAA Paper 200-4369. 2000.

11. Moon, T.K. and Stirling, W.C. *Mathematical Methods and Algorithms*. New Jersey: Prentice Hall, 2000.

12. Eppstein, D. "Finding the *k* Shortest Paths" March 1997.

13. Novy, M.C. and Jacques, D.R. "Air Vehicle Optimal Trajectories Between Two Radars" Proceedings of the *American Control Conference*, Anchorage, AK, May 2002.

14. Judd, K.B., and McLain, T.W. "Spline Based Path Planning for Unmanned Air Vehicles" AIAA Paper 2001-4238.

15. Herbert, J., Jacques, D., Novy, M., and Pachter, M. "Cooperative Control of UAVs" AIAA Paper 2001-4240. AIAA *Guidance, Navigations, and Control Conference,* Montreal, Canada, August 2001.

16. Anderson, E.P., and Beard, R.W. "An Algorithmic Implementation of Constrained Extremal Control for UAVs" AIAA Paper 2002-4470. AIAA *Guidance, Navigations, and Control Conference,* Monterey, CA, August 2002.

17. Chen, G., and Cruz, J.B. "Genetic Algorithm for Task Allocation in UAV Cooperative Control" AIAA Paper 2003-5582. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

18. Bellingham, J., Tillerson, M., Richards, A., How, J. "Multi-Task Allocation and Trajectory Design for Cooperating UAVs," in Cooperative Control: Models, Applications and Algorithms at the *Conference on Coordination, Control and Optimization*, November 2001.

19. Moser, M., Jokanovic, D.P., and Shiratori, N. "An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem" *IEICE Trans. Fundamentals,* Vol. E80-A, No. 3, March 1997.

20. Richards, A., Bellingham, J., Tillerson, M., How, J. "Co-ordination and Control of Multiple UAVs" AIAA *Guidance, Navigation, and Control Conference*, Monterey, CA, August 2002.

21. Richards, A, M., How, J. "Aircraft Trajectory Planning with Collision Avoidance Using Mixed Integer Linear Programming" Proceedings of the *American Control Conference*, Anchorage, AK, May 2002.

22. Schouwenaars, T., De Moor, B., Feron, E., and How, J. "Mixed Integer Linear Programming for Multi-Vehicle Path Planning" ECC Conference, 2001.

23. Richards, A., Kuwata, Y., and How, J. "Experimental Demonstrations of Real-time MILP Control" AIAA Paper 2003-5802. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

24. Chandler, P.R., Pachter, M., Rasmussen, S., and Schumacher, C. "Distributed Control for Multiple UAVs with Strongly Coupled Tasks" Paper 2003-5799. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

25. Chandler, P.R., and Pachter, M., "Hierarchical Control for Autonomous Teams" AIAA Paper 2001-4149. AIAA *Guidance, Navigations, and Control Conference,* Montreal, Canada, August 2001.

26. Chandler, P.R., Pachter, M., Swaroop, D., Fowler, J.M., Howlett, J.K., Rasmussen, S., Schumacher, C., Nygard, K., "Complexity in UAV Cooperative Control" Proceedings of the *American Control Conference*, Anchorage, AK, May 2002.

27. Boskovic, J.D., Prasanth, R., and Mehra, R.K. "An Autonomous Hierarchical Control Architecture for Unmanned Aerial Vehicles" AIAA Paper 2002-4468. AIAA *Guidance, Navigations, and Control Conference,* Monterey, CA, August 2002.

28. Boskovic, J.D., Prasanth, R., and Mehra, R.K. "A Multi-Layer Control Architecture for Unmanned Aerial Vehicles" Proceedings of the *American Control Conference*, Anchorage, AK, May 2002.

29. Howlett, J.K. "Path Planning and Cooperative Assignment" Air Vehicles Directorate, Wright-Patterson Air Force Base, Ohio, Summer 2001.

30. Verma, A, Wu, C., and Castelli, V. "Autonomous Command and Control for UAV Formation" Paper 2003-5704. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

31. Schumacher, C., Chandler, P.R., and Rasmussen, S. "Task Allocation for a Wide Area Search Munition via Iterative Network Flow" AIAA Paper 2002-4586. AIAA *Guidance, Navigations, and Control Conference,* Monterey, CA, August 2002.

32. Schumacher, C., Chandler, P.R., Pachter, M., and Pachter, L.S. "UAV Task Assignment with Timing Constraints" Paper 2003-5664. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

33. Rasmussen, S., Chandler, P., Mitchell, J.W., Schumacher, C., and Sparks, A. "Optimal vs. Heuristic Assignment of Cooperative Autonomous Unmanned Air Vehicles" Paper 2003-5586. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

34. Rasmussen, S., Mitchell, J.W., Schulz, C., Schumacher, C., and Chandler, P. "A Multiple UAV Simulation for Researchers" Paper 2003-5684. AIAA *Guidance, Navigations, and Control Conference,* Austin, TX, August 2003.

35. Carpenter, J.R. "Partially Decentralized Control Architectures for Satellite Formations" AIAA Paper 2002-4959. AIAA *Guidance, Navigations, and Control Conference,* Monterey, CA, August 2002.

36. Boskovic, J.D., and Mehra, R.K. "A Decentralized Scheme for Autonomous Compensation of Multiple Simultaneous Flight-Critical Failures" AIAA Paper

2002-4453. AIAA *Guidance, Navigations, and Control Conference*, Monterey, CA, August 2002.

37. Yang, Y., Minai, A.A., Polycarpou, M.M., "Decentralized Cooperative Search in UAVs Using Opportunistic Learning" AIAA Paper 2002-4590. AIAA *Guidance, Navigations, and Control Conference*, Monterey, CA, August 2002.

38. Mitchell, J.W., Schumacher, C., Chandler, P.R., "Communication Delays in the Cooperative Control of Wide Area Search Munitions Via Iterative Network" AIAA Paper 2003-5665. AIAA *Guidance, Navigation, and Control Conference*, Austin, TX, August 2003.

39. Ashokkumar, C.R., and Jeffcoat, D.E., "Cooperative Systems Under Communication Delay" AIAA Paper 2003-5663. AIAA *Guidance, Navigations, and Control Conference*, Austin, TX, August 2003.

40. de Berg, M., van Kreveld, M., Schwarzkopf, O., and Overmarr, M. *Computational Geometry: Algorithms and Applications, Second Edition.* New York: Springer-Verlag, 2000.

41. Global Security.org "World Military Guide". http://www.globalsecurity.org/military/world/index.html

42. Wikipedia. "Dijkstra's Algorithm" http://en.wikipedia.org/wiki/Dikjstra's_algorithm

43. Wikipedia. "Adjacency Matrix" http://en.wikipedia.org/wiki/Advacency_matrix

44. Kay, Michael. MATLOG, MATLAB toolbox package. Available from http://www.ie.ncsu.edu/kay/matlog

45. Wikipedia. "NP-hard" http://en.wikipedia.org/wiki/NP-hard

46. Akbar, M.M., Manning, E.G., Shoja, G.C., and Khan, S. "Heuristic Solution for the Multiple-Choice Multi-Dimensional Knapsack Problem" International Conference on Computational Science, San Francisco, May 2001.

47. Roskam, J. *Airplane Fight Dynamics and Automatic Flight Controls: Part I.* DARcorporation: Lawrence, 1995.

48. Stevens, B.L., and Lewis, F.L. *Aircraft Control and Simulation.* John Wiley and Sons: New York, 1992.

49. Rauw, M. *FDC 1.2 – A SIMULINK Toolbox for Flight Dynamics and Control Analysis.* May, 2001. http://home.wanadoo.nl/dutchroll/manual.html

Appendix A

MATLAB Codes for Path Planning and Task Allocation

path_planning.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [out]=path_planning(in)

UAVS_long=in([1:36],1);
UAVS_long=reshape(UAVS_long,4,9);
TARGETS_long=in([37:72]);
TARGETS_long=reshape(TARGETS_long,4,9);
ZONES_long=in([73:102]);
ZONES_long=reshape(ZONES_long,3,10);
THREATS_long=in([103:162]);
THREATS_long=reshape(THREATS_long,4,15);
TIME=in(163);
n_plots=in(164);
HEADING_ANGLE=in([165:173]);

uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS_long(:,i)))>0 & abs(sum(UAVS_long(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end
[UAVS]=filter_zeros(UAVS_long,9);
n_uav=size(UAVS,2);

targ_existing=zeros(1,9);
for i=1:9
   if TARGETS_long(3,i)~=0,
      targ_existing(1,i)=1;
  end
end
[TARGETS_temp]=filter_zeros(TARGETS_long,9);
TARGETS=[TARGETS_temp(1,:);TARGETS_temp(2,:)];
n_targ=size(TARGETS,2);

[ZONES]=filter_zeros(ZONES_long,10);
n_zones=size(ZONES,2);

threats_existing=zeros(1,15);
for i=1:15
   if THREATS_long(3,i)~=0
      threats_existing(1,i)=1;
  end
end
[THREATS]=filter_zeros(THREATS_long,15);
n_threats=size(THREATS,2);

ZONES_REAL=ZONES;
THREATS_REAL=THREATS;

ZONES(3,:)=1.15*ZONES_REAL(3,:);
THREATS(3,:)=1.15*THREATS_REAL(3,:);

split_seg=10;
min_turn=1;
[all_pos,all_lines_x,all_lines_y,all_costs]=vrn_diag_gen(UAVS,TARGETS,ZONES,THREATS);
```

```
[stored_paths,totalcost]=cheapest_paths(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZONE
S,THREATS);
[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng(stored_paths,all_pos,ZONES,THREATS,m
in_turn,split_seg,n_uav,n_targ,HEADING_ANGLE);
[Selected_Paths_x,Selected_Paths_y]=mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Path
s_y,n_uav);
[uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(Selected_Paths_x,Selected_Paths_y,UA
VS,min_turn*2);
if n_plots~=0,

plot_uav(UAVS_long,TARGETS_long,ZONES_REAL,THREATS_long,uav_path_x,uav_path_y,n_plots,
uavs_existing,targ_existing,threats_existing);
end

disp(sprintf('Path Planning ran at time %d. \n',round(TIME)));

bestcomb=zeros(1,9);
for i=1:n_uav,
   for j=1:n_targ,
      if round(Selected_Paths_x(end,i)*10)==round(TARGETS(1,j)*10) &
            round(Selected_Paths_y(end,i)*10)==round(TARGETS(2,j)*10)
         bestcomb(1,i)=j;
         break
      end
   end
end

%Making into vector
uav_x=zeros(9,100);
uav_y=zeros(9,100);
uav_time=zeros(9,100);
uav_alt=zeros(9,100);
selected_targets=zeros(9,1);
szpath=size(uav_path_x,2);
counter=1;
for i=1:9,
   if uavs_existing(1,i)==1
      selected_targets(i,1)=bestcomb(1,counter);
      uav_x(i,[1:szpath])=uav_path_x(counter,:);
      uav_y(i,[1:szpath])=uav_path_y(counter,:);
      uav_time(i,[1:szpath])=time_uav(counter,:)+TIME;
      uav_alt(i,[1:szpath])=altitude_uav(counter,:);
      counter=counter+1;
   end
end
sys_temp=[ ];
for i=1:9;
   sys_temp=[sys_temp,uav_x(i,:),uav_y(i,:),uav_alt(i,:),uav_time(i,:)];
end
out=[sys_temp,selected_targets'];
```

filter_zeros.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [A]=filter_zeros(A_long,n)

A=[ ];
counter=1;
for i=1:n
    if abs(sum(A_long(:,i)))>0 & abs(sum(A_long(:,i)))~=0.26
        A(:,counter)=A_long(:,i);
        counter=counter+1;
   end
end
```

v***rn_diag_gen.m***

Authored by Matthew Lechliter, Zachary Spritzer, and Jennifer Hazelton

```
function [all_pos,all_lines_x,all_lines_y,all_costs]=vrn_diag_gen(UAVS,TARGETS,ZONES,THREATS)

%INPUTS:
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%TARGETS - is a 2xn matrix where n is the number of Targets, the first row
%is the x position of the targets and the second row is the y position of
%the targets.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y

max_x=max([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])+25;
min_x=min([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])-25;
max_y=max([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])+25;
min_y=min([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])-25;

VRNPTS=[ZONES([1,2],:) THREATS([1,2],:) ...
   [(((max_y-min_y)*[1:4]/4)+min_y);(min_x)*ones(1,4)] ...
   [(((max_y-min_y)*[1:4]/4)+min_y);(min_x)*ones(1,4)] ...
   [(((max_x-min_x)*[1:4]/4)+min_x);(min_y)*ones(1,4)] ...
   [(((max_x-min_x)*[1:4]/4)+min_x);(max_y)*ones(1,4)]];
```

```
[vx,vy] = voronoi(VRNPTS(1,:),VRNPTS(2,:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Taking unique numbers from vx and vy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[vxyn]= 1e-6*unique(round(1e6*[vx(:),vy(:)]),'rows');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting UAV's into voronoi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_uav,uavx,uavy]=connect_vrn(vxyn,UAVS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting the targets into the voronoi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_targ,targx,targy]=connect_vrn(vxyn,TARGETS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generation for voronoi line costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nvlines=size(vx,2);
line_cost_vrn=zeros(1,nvlines);
for i=1:nvlines,
    line_cost_vrn(1,i)=sqrt((vx(1,i)-vx(2,i))^2+(vy(1,i)-vy(2,i))^2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Stacking unique positions, lines for x and y, and costs of those lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
all_pos=[UAVS([1,2],:) vxyn(:,[1,2])' TARGETS([1,2],:)];
all_lines_x=[uavx([1,2],:) vx([1,2],:) targx([1,2],:)];
all_lines_y=[uavy([1,2],:) vy([1,2],:) targy([1,2],:)];
all_costs=[line_cost_uav(1,:) line_cost_vrn(1,:) line_cost_targ(1,:)];
```

voronoi.m

```
function [vxx,vy] = voronoi(x,y,arg3,arg4)
%VORONOI Voronoi diagram.
%   VORONOI(X,Y) plots the Voronoi diagram for the points X,Y.
%   Cells that contain a point at infinity are unbounded and
%   are not plotted.
%
%   VORONOI(X,Y,TRI) uses the triangulation TRI instead of
%   computing it via DELAUNAY.
%
%   H = VORONOI(...,'LineSpec') plots the diagram with color and linestyle
%   specified and returns handles to the line objects created in H.
%
%   [VX,VY] = VORONOI(...) returns the vertices of the Voronoi
%   edges in VX and VY so that plot(VX,VY,'-',X,Y,'.') creates the
%   Voronoi diagram.
%
%   For the topology of the voronoi diagram, i.e. the vertices for
%   each voronoi cell, use the function VORONOIN as follows:
%
%       [V,C] = VORONOIN([X(:) Y(:)])
%
%   See also VORONOIN, DELAUNAY, CONVHULL.

%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 1.15 $  $Date: 2002/06/05 20:05:17 $

error(nargchk(2,4,nargin));

if nargin==2,
  tri = delaunay(x,y);
  ls = '';
elseif nargin==3,
  if isstr(arg3),
    tri = delaunay(x,y);
    ls = arg3;
  else
    tri = arg3;
    ls = '';
  end
else
  tri = arg3;
  ls = arg4;
end

% re-orient the triangles so that they are all clockwise
xt = x(tri); yt=y(tri);
ot = xt(:,1).*(yt(:,2)-yt(:,3)) + ...
     xt(:,2).*(yt(:,3)-yt(:,1)) + ...
     xt(:,3).*(yt(:,1)-yt(:,2));
bt = find(ot<0);
tri(bt,[1 2]) = tri(bt,[2 1]);

n = prod(size(x));
ntri = size(tri,1);
```

```
t = (1:ntri)';
T = sparse(tri,tri(:,[3 1 2]),t(:,ones(1,3)),n,n); % Triangle edge if T(i,j)
E = (T & T').*T; % Voronoi edge if E(i,j)

[i,j,v] = find(triu(E));
[i,j,vv] = find(triu(E'));
c1 = circle(tri(v,:),x,y);
c2 = circle(tri(vv,:),x,y);

vx = [c1(:,1) c2(:,1)].';
vy = [c1(:,2) c2(:,2)].';

if nargout<2
  if isempty(ls),
    co = get(gcf,'defaultaxescolororder');
    h = plot(vx,vy,'-',x,y,'.','color',co(1,:));
  else
    [l,c,m,msg] = colstyle(ls); error(msg)
    if isempty(m), m = '.'; end
    h = plot(vx,vy,ls,x,y,[c m]);
  end
  if ~ishold,
    view(2), axis([min(x(:)) max(x(:)) min(y(:)) max(y(:))])
  end
  if nargout==1, vxx = h; end
else
  vxx = vx;
end

function c = circle(tri,x,y)
%CIRCLE Return center and radius for circumcircles
%   C = CIRCLE(TRI,X,Y) returns a N-by-3 vector containing [xcenter(:)
%   ycenter(:) radius(:)] for each triangle in TRI.

% Reference: Watson, p32.
x = x(:); y = y(:);

x1 = x(tri(:,1)); x2 = x(tri(:,2)); x3 = x(tri(:,3));
y1 = y(tri(:,1)); y2 = y(tri(:,2)); y3 = y(tri(:,3));

% Set equation for center of each circumcircle:
%    [a11 a12;a21 a22]*[x;y] = [b1;b2] * 0.5;

a11 = x2-x1; a12 = y2-y1;
a21 = x3-x1; a22 = y3-y1;

b1 = a11 .* (x2+x1) + a12 .* (y2+y1);
b2 = a21 .* (x3+x1) + a22 .* (y3+y1);

% Solve the 2-by-2 equation explicitly
idet = a11.*a22 - a21.*a12;

% Add small random displacement to points that are either the same
% or on a line.
d = find(idet == 0);
if ~isempty(d), % Add small random displacement to points
```

```
  delta = sqrt(eps);
  x1(d) = x1(d) + delta*(rand(size(d))-0.5);
  x2(d) = x2(d) + delta*(rand(size(d))-0.5);
  x3(d) = x3(d) + delta*(rand(size(d))-0.5);
  y1(d) = y1(d) + delta*(rand(size(d))-0.5);
  y2(d) = y2(d) + delta*(rand(size(d))-0.5);
  y3(d) = y3(d) + delta*(rand(size(d))-0.5);
  a11 = x2-x1; a12 = y2-y1;
  a21 = x3-x1; a22 = y3-y1;
  b1 = a11 .* (x2+x1) + a12 .* (y2+y1);
  b2 = a21 .* (x3+x1) + a22 .* (y3+y1);
  idet = a11.*a22 - a21.*a12;
end

idet = 0.5 ./ idet;

xcenter = ( a22.*b1 - a12.*b2) .* idet;
ycenter = (-a21.*b1 + a11.*b2) .* idet;

radius = (x1-xcenter).^2 + (y1-ycenter).^2;

c = [xcenter ycenter radius];
```

connect_vrn.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [line_cost_uav,uavx,uavy]=connect_vrn(vxyn,UAVS)

%Inputs:
%
%vxyn - is a nx2 matrix with first column defining all of the unique x
%positions of the voronoi diagram or grid and the second column defining
%all of the unique y positions of the voronoi diagram or grid.
%
%UAVS - is a 2xn matrix with the first row defining the x position of the
%UAV and the second row defining the y position of the UAV.
%
%Outputs:
%
%line_cost_uav - is a vector containing the cost of the lines of connecting
%the UAV's into the voronoi diagram or grid
%
%uavx - is a 2xn matrix with first row defining ending point and second row
%defining starting point for the x coordinates.
%
%uavy - is a 2xn matrix with first row defining ending point and second row
%defining starting point for the y coordinates.
nuav=size(UAVS,2);
nvxynpts=size(vxyn,1);
du=zeros(1,nvxynpts-1);
uavx=zeros(2,nuav*3);
uavy=zeros(2,nuav*3);
line_cost_uav=zeros(1,nuav*3);
for k=1:nuav,
   for j=2:nvxynpts,
      du(1,j-1)=sqrt((UAVS(1,k)-vxyn(j,1))^2+(UAVS(2,k)-vxyn(j,2))^2);
   end
   mdu=sort(du,2);
   for i=1:3,
      mdu_loc=find(du==mdu(1,i));
      uavx(1,3*(k-1)+i)=vxyn(mdu_loc+1,1);
      uavy(1,3*(k-1)+i)=vxyn(mdu_loc+1,2);
      uavx(2,3*(k-1)+i)=UAVS(1,k);
      uavy(2,3*(k-1)+i)=UAVS(2,k);
      line_cost_uav(1,3*(k-1)+i)=mdu(1,i);
   end
end
```

cheapest_paths.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function
[stored_paths,totalcost]=cheapest_paths(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZONE
S,THREATS)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y.
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%TARGETS - is a 2xn matrix where n is the number of Targets, the first row
%is the x position of the targets and the second row is the y position of
%the targets.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%stored_paths - is a mxn matrix where m is the number of uavs times the
%number of targets and n is the length of the longest path.  The first row
%being the first path for the first uav and the last row being the last
%path for the last uav. The paths are output by node numbers coming from
%the implementation of dijkstra's algorithm.
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Making THC matrix for dijkstra's algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[THC]=set_THC(all_pos,all_lines_x,all_lines_y,all_costs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Cost Assignment for all lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[THC]= c_assign(all_pos,THC,ZONES,THREATS);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding the reverse of the THC matrix onto the end, so that the
%reverse of the lines is possible
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
THC=[THC(:,[1,2,3]); THC(:,[2,1,3])];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Implementing Dijkstra's algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nuav=size(UAVS,2);
ntarg=size(TARGETS,2);
A = list2adj(THC);
totalcost=zeros(nuav,ntarg);
for i=1:nuav,
    for j=1:ntarg,
        [totalcost(i,j),path] = dijk(A,i,size(all_pos,2) - j + 1);
        stored_paths((i-1)*ntarg+j,[1:size(path,2)])=path(1,[1:size(path,2)]);
    end
end
```

set_THC.m

Authored by Matthew Lechliter, Zachary Spritzer, and Elena Lucci

```
function [THC]=set_THC(all_pos,all_lines_x,all_lines_y,all_costs)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y.
%
%OUTPUTS:
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line. With updated costs due to no-fly zones and
%threats.



THC=zeros(size(all_lines_x,2),3);
THC(:,3)=all_costs(:);
for i=1:(2*size(all_lines_x,2))
   P=(round(all_pos(1,:)*100)== round(all_lines_x(i)*100)) &
(round(all_pos(2,:)*100)==round(all_lines_y(i)*100));
   if  any(P)
      num=find(P);
      if (rem(i,2))~=0
         bz=((fix(i./2))+1);
         THC(bz,1)=num;
      else THC((i/2),2)=num;
      end
   else
      if (rem(i,2))~=0
         tz=(fix((i./2))+1);
         THC(tz,1)=i;
      else THC((i/2),2)=i;
      end
   end
end
```

c_assign.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [THC]= c_assign(all_pos,THC,ZONES,THREATS)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line. With updated costs due to no-fly zones and
%threats.
szthc=size(THC,1);
nzones=size(ZONES,2);
nthrts=size(THREATS,2);

for i=1:szthc,
   start=THC(i,1);finish=THC(i,2);
   SF=sqrt(((all_pos(1,finish)-all_pos(1,start))^2)+((all_pos(2,finish)-all_pos(2,start))^2));
   for j=1:nzones,
      SC=sqrt(((ZONES(1,j)-all_pos(1,start))^2)+((ZONES(2,j)-all_pos(2,start))^2));
      FC=sqrt(((ZONES(1,j)-all_pos(1,finish))^2)+((ZONES(2,j)-all_pos(2,finish))^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
      else
         if SC<FC,PC=SC;
         else
            PC=FC;
         end
      end
      if PC < ZONES(3,j),THC(i,3)=1e30*THC(i,3);
      end
   end
   for j=1:nthrts,
      SC=sqrt(((THREATS(1,j)-all_pos(1,start))^2)+((THREATS(2,j)-all_pos(2,start))^2));
      FC=sqrt(((THREATS(1,j)-all_pos(1,finish))^2)+((THREATS(2,j)-all_pos(2,finish))^2));
```

```
        SN=(SC^2+SF^2-FC^2)/(2*SF);
        if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
        else
            if SC<FC,PC=SC;
            else
                PC=FC;
            end
        end
        if PC < THREATS(3,j),THC(i,3)=(THREATS(4,j)*100)+THC(i,3);
        end
    end
end
```

list2adj.m

```
function A = list2adj(IJC,m,spA)
%LIST2ADJ Arc list to node-node weighted adjacency matrix representation.
%     A = list2adj(IJC,m,spA)
%   IJC = n x 2-5 matrix arc list [i j c u l], where
%     i = n-element vector of arc tails nodes
%     j = n-element vector of arc head nodes
%     c = (optional) n-element vector of arc costs, where n = number of arcs
%       = (default) ONES(n,1)
%     u = (optional) ignored
%     l = (optional) ignored
%     m = (optional) scalar size of A if greater than max{max(i),max(abs(j))}
%   spA = (optional) make A sparse matrix if n <= spA x m x m
%       = 1, always make A sparse
%       = 0.1 (default), A sparse if 10% arc density
%       = 0, always make A full matrix
%     A = m x m node-node weighted adjacency matrix
%
% Transforms: If j(k) > 0, then [i(k) j(k) c(k)] -> A[i(k),j(k)]  = c(k)
%             If j(k) < 0, then [i(k) j(k) c(k)] -> A[i(k),-j(k)] = c(k) and
%                                                   A[-j(k),i(k)] = c(k)
%
% Note: Weights of any duplicate arcs added together in A
%       c(k) = 0 => A(i(k),j(k)) = NaN
%       Wrapper for c(c==0) = NaN; A = SPARSE(i,j,c,m,m);
%
% See also LIST2INCID, ADJ2LIST, and ADJ2INCID

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
error(nargchk(1,3,nargin))

[n,cIJC] = size(IJC);
if cIJC < 2 | cIJC > 5, error('IJC must be a 2-3 column matrix.'), end

[i,j,c] = mat2vec(IJC);
if isempty(c), c = ones(n,1); end

jsgn = sign(j); j = abs(j);
minIJ = min(min([i j]));
if isempty(minIJ) | minIJ < 1 | any(~isint(i)) | any(~isint(j))
   error('All elements of "i" and "j" must be nonzero integers.');
end

if nargin < 2 | isempty(m)
   m = max(max([i j]));
elseif length(m(:)) ~= 1 | ~isint(m) | m < max(max([i j]))
   error('"n" must be >= max{max(i),max(abs(j))}.');
end

if nargin < 3 | isempty(spA)
   spA = 0.1;
elseif length(spA(:)) ~= 1 | spA < 0
```

```
  error('"spA" must be non-negative scalar.');
end
% End (Input Error Checking) ************************************************

if any(jsgn < 0)                                        % Add elements from undirected arcs
  jsgn(jsgn < 0 & i == j) = 1;
  i = [i; j(jsgn < 0)];
  j = [j; i(jsgn < 0)];
  c = [c; c(jsgn < 0)];
end

c(c==0) = NaN;
A = sparse(i,j,c,m,m);

if n > spA * m * m, A = full(A); end
```

adj2list.m

```
function [i,j,c] = adj2list(A)
%ADJ2LIST Node-node weighted adjacency matrix to arc list representation.
%     IJC = adj2list(A)
% [i,j,c] = adj2list(A)
%     A = m x m node-node weighted adjacency matrix of arc lengths
%   IJC = n x 2-3 matrix arc list [i j c], where
%     i = n-element vector of arc tails nodes
%     j = n-element vector of arc head nodes
%     c = n-element vector of arc weights
%
% Note: All A(i,j) = A(j,i) => [i -j c] (symmetric A)
%       A(i,j) = 0   => Arc (i,j) does not exist
%       A(i,j) = NaN => Arc (i,j) exists with 0 weight
%       Wrapper for [i,j,c] = FIND(C); c(ISNAN(c)) = 0)
%
% See also LIST2INCID, LIST2ADJ, and ADJ2INCID

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
[rA,cA] = size(A);
if rA ~= cA
   error('"A" must be a square matrix.');
end
% End (Input Error Checking) ************************************************

if all(all(triu(A)==tril(A)')), A = triu(A); issym = 1; else issym = 0; end

[i,j,c] = find(A);
if issym, j = -j; end
c(isnan(c)) = 0;

if nargout == 1
   i = [i j c];
end
```

pred2path.m

```
function rte = pred2path(P,s,t)
%PRED2PATH Convert predecessor indices to shortest paths from node 's' to 't'.
%   rte = pred2path(P,s,t)
%     P = |s| x n matrix of predecessor indices (from DIJK)
%     s = FROM node indices
%       = [] (default), paths from all nodes
%     t = TO node indices
%       = [] (default), paths to all nodes
%   rte = |s| x |t| cell array of paths (or routes) from 's' to 't', where
%         rte{i,j} = path from s(i) to t(j)
%                  = [], if no path exists from s(i) to t(j)
%
% (Used with output of DIJK)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
error(nargchk(1,3,nargin));

[rP,n] = size(P);

if nargin < 2 | isempty(s), s = (1:n)'; else s = s(:); end
if nargin < 3 | isempty(t), t = (1:n)'; else t = t(:); end

if any(P < 0 | P > n)
   error(['Elements of P must be integers between 1 and ',num2str(n)]);
elseif any(s < 1 | s > n)
   error(['"s" must be an integer between 1 and ',num2str(n)]);
elseif any(t < 1 | t > n)
   error(['"t" must be an integer between 1 and ',num2str(n)]);
end
% End (Input Error Checking) ************************************************

rte = cell(length(s),length(t));

[ans,idxs] = find(P==0);

for i = 1:length(s)
%    if rP == 1
%       si = 1;
%    else
%       si = s(i);
%       if si < 1 | si > rP
%          error('Invalid P matrix.')
%       end
%    end
   si = find(idxs == s(i));
   for j = 1:length(t)
      tj = t(j);
      if tj == s(i)
         r = tj;
      elseif P(si,tj) == 0
         r = [];
```

```
    else
      r = tj;
      while tj ~= 0
        if tj < 1 | tj > n
          error('Invalid element of P matrix found.')
        end
        r = [P(si,tj) r];
        tj = P(si,tj);
      end
      r(1) = [];
    end
    rte{i,j} = r;
  end
end

if length(s) == 1 & length(t) == 1
  rte = rte{:};
end

%rte = t;
while 0%t ~= s
  if t < 1 | t > n | round(t) ~= t
    error('Invalid "pred" element found prior to reaching "s"');
  end
  rte = [P(t) rte];
  t = P(t);
end
```

mat2vec.m

```
function varargout = mat2vec(X)
%MAT2VEC Convert columns of matrix to vectors.
% [X(:,1),X(:,2),...] = mat2vec(X)
%
% (Additional output vectors assigned as empty)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
if ~isnumeric(X)
   error('X must be numeric.')
end
% End (Input Error Checking) ************************************************

varargout = cell(1,max(1,nargout));
X = num2cell(X,1);
varargout(1,1:min(nargout,size(X,2))) = X(1,1:min(nargout,size(X,2)));
```

isint.m

```
function y = isint(x,TolInt)
%ISINT True for integer elements (within tolerance).
%      y = isint(x,TolInt)
%        = abs(x-round(x)) < TolInt
% TolInt = integer tolerance
%        = [0.01*sqrt(eps)], default

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
error(nargchk(1,2,nargin));
if nargin < 2 | isempty(TolInt), TolInt = 0.01*sqrt(eps); end
% End (Input Error Checking) ************************************************

y = abs(x-round(x)) < TolInt;
```

dijk.m

```
function [D,P] = dijk(A,s,t)
%DIJK Shortest paths from nodes 's' to nodes 't' using Dijkstra algorithm.
% [D,P] = dijk(A,s,t)
%     A = n x n node-node weighted adjacency matrix of arc lengths
%         (Note: A(i,j) = 0   => Arc (i,j) does not exist;
%                A(i,j) = NaN => Arc (i,j) exists with 0 weight)
%     s = FROM node indices
%       = [] (default), paths from all nodes
%     t = TO node indices
%       = [] (default), paths to all nodes
%     D = |s| x |t| matrix of shortest path distances from 's' to 't'
%       = [D(i,j)], where D(i,j) = distance from node 'i' to node 'j'
%     P = |s| x n matrix of predecessor indices, where P(i,j) is the
%         index of the predecessor to node 'j' on the path from 's(i)' to
%         'j',where P(i,i) = 0 and P(i,j) = NaN is 'j' not on path to 's(i)'
%         (use PRED2PATH to convert P to paths)
%       = path from 's' to 't', if |s| = |t| = 1
%
%  (If A is a triangular matrix, then computationally intensive node
%   selection step not needed since graph is acyclic (triangularity is a
%   sufficient, but not a necessary, condition for a graph to be acyclic)
%   and A can have non-negative elements)
%
%  (If |s| >> |t|, then DIJK is faster if DIJK(A',t,s) used, where D is now
%   transposed and P now represents successor indices)
%
%  (Based on Fig. 4.6 in Ahuja, Magnanti, and Orlin, Network Flows,
%   Prentice-Hall, 1993, p. 109.)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
error(nargchk(1,3,nargin))

[n,cA] = size(A);

if nargin < 2 | isempty(s), s = (1:n)'; else s = s(:); end
if nargin < 3 | isempty(t), t = (1:n)'; else t = t(:); end

if ~any(any(tril(A) ~= 0))       % A is upper triangular
   isAcyclic = 1;
elseif ~any(any(triu(A) ~= 0))   % A is lower triangular
   isAcyclic = 2;
else                             % Graph may not be acyclic
   isAcyclic = 0;
end

if n ~= cA
   error('A must be a square matrix');
elseif ~isAcyclic & any(any(A < 0))
   error('A must be non-negative');
elseif any(s < 1 | s > n)
   error(['"s" must be an integer between 1 and ',num2str(n)]);
```

```
elseif any(t < 1 | t > n)
  error(['"t" must be an integer between 1 and ',num2str(n)]);
end
% End (Input Error Checking) ************************************************

A = A';    % Use transpose to speed-up FIND for sparse A

D = zeros(length(s),length(t));
if nargout > 1, P = NaN*ones(length(s),n); end

for i = 1:length(s)
  j = s(i);

  Di = Inf*ones(n,1); Di(j) = 0;

  isLab = logical(zeros(length(t),1));
  if isAcyclic ==  1
    nLab = j - 1;
  elseif isAcyclic == 2
    nLab = n - j;
  else
    nLab = 0;
    UnLab = 1:n;
    isUnLab = logical(ones(n,1));
  end

  if nargout > 1, P(i,s(i)) = 0; end  % Change from NaN to indicate no pred

  while nLab < n & ~all(isLab)
    if isAcyclic
      Dj = Di(j);
    else  % Node selection
      [Dj,jj] = min(Di(isUnLab));
      j = UnLab(jj);
      UnLab(jj) = [];
      isUnLab(j) = 0;
    end

    nLab = nLab + 1;
    if length(t) < n, isLab = isLab | (j == t); end

    [jA,kA,Aj] = find(A(:,j));
    Aj(isnan(Aj)) = 0;

    if isempty(Aj), Dk = Inf; else Dk = Dj + Aj; end

    if nargout > 1, P(i,jA(Dk < Di(jA))) = j; end
    Di(jA) = min(Di(jA),Dk);

    if isAcyclic == 1       % Increment node index for upper triangular A
      j = j + 1;
    elseif isAcyclic == 2   % Decrement node index for lower triangular A
      j = j - 1;
    end
  end
  D(i,:) = Di(t)';
```

```
end

if nargout > 1 & length(s) == 1 & length(t) == 1
   P = pred2path(P,s,t);
end
```

path_shrtng.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [Shortened_Paths_x,Shortened_Paths_y,totalcost]=
path_shrtng(stored_paths,all_pos,ZONES,THREATS,min_turn,split_seg,nuav,ntarg,HEADING_ANGLE)

%INPUTS:
%
%stored_paths - is a mxn matrix where m is the number of uavs times the
%number of targets and n is the length of the longest path.  The first row
%being the first path for the first uav and the last row being the last
%path for the last uav. The paths are output by node numbers coming from
%the implementation of dijkstra's algorithm.
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%min_turn - minimum turning radius for the UAVs
%
%split_seg - number of segments to Split the voronoi lines into for the
%purpose of a more near-optimal solution
%
%nuav - number of UAVs
%
%ntarg - number of targets

%OUTPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
%
%Stored_Pos - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Splitting the voronoi lines into more segments for the purpose of a more near-optimal solution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
szpths=size(stored_paths,2);

split_vect=[(0:(1/split_seg):(1- 1/split_seg))]';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Finding the corresponding x and y coordinates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stored_Pos_x=ones(szpths,nuav*ntarg);
Stored_Pos_y=ones(szpths,nuav*ntarg);
stored_paths(:,szpths+1)=0;
for i=1:nuav*ntarg,
   mnz=min(find(stored_paths(i,:)==0));
   Stored_Pos_x(1:mnz-1,i)=all_pos(1,stored_paths(i,1:mnz-1))';
   Stored_Pos_y(1:mnz-1,i)=all_pos(2,stored_paths(i,1:mnz-1))';
   Stored_Pos_x(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(1,stored_paths(i,mnz-1))';
   Stored_Pos_y(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(2,stored_paths(i,mnz-1))';
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stored_Pos_x_new=ones((((szpths-1)*split_seg)+1),nuav*ntarg);
Stored_Pos_y_new=ones((((szpths-1)*split_seg)+1),nuav*ntarg);
for k=1:nuav*ntarg,
      j=1;
   for i=1:(szpths -1),
        Stored_Pos_x_new([j:(j + (split_seg -1))],k)=
ones(split_seg,1)*Stored_Pos_x(i,k)+split_vect*(Stored_Pos_x(i+1,k)-Stored_Pos_x(i,k));
        Stored_Pos_y_new([j:(j + (split_seg -1))],k)=
ones(split_seg,1)*Stored_Pos_y(i,k)+split_vect*(Stored_Pos_y(i+1,k)-Stored_Pos_y(i,k));
        j=j+ split_seg;
   end
   Stored_Pos_x_new((((szpths-1)*split_seg)+1),k)=Stored_Pos_x(szpths,k);
   Stored_Pos_y_new((((szpths-1)*split_seg)+1),k)=Stored_Pos_y(szpths,k);
end

Shortened_Paths_x_end=ones(500,1)*Stored_Pos_x(szpths,:);
Shortened_Paths_y_end=ones(500,1)*Stored_Pos_y(szpths,:);
Shortened_Paths_x=[Stored_Pos_x_new;Shortened_Paths_x_end];
Shortened_Paths_y=[Stored_Pos_y_new;Shortened_Paths_y_end];
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Shortening the paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav*ntarg,

[Shortened_Paths_x(:,i),Shortened_Paths_y(:,i)]=shorten_paths(Shortened_Paths_x(:,i),Shortened_Paths_y
(:,i),ZONES,THREATS,Stored_Pos_x(:,i),Stored_Pos_y(:,i));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Putting fillets into the shortened paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav*ntarg,

[Shortened_Paths_x(:,i),Shortened_Paths_y(:,i)]=fillet_path([Shortened_Paths_x(:,i),Shortened_Paths_y(:,i
)],min_turn);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding initial path based on heading angle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav,
    for j=1:ntarg,
        [Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-1)*ntarg)+j)]=...
            heading_angle_paths([Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-
1)*ntarg)+j)],min_turn,HEADING_ANGLE(i,1),72);
    end
end


Shortened_Paths_x_old=Shortened_Paths_x;
Shortened_Paths_y_old=Shortened_Paths_y;
Shortened_Paths_x=[];
Shortened_Paths_y=[];
for j=1:size(Shortened_Paths_x_old,1)-1,
    if Shortened_Paths_x_old(j,:)==Shortened_Paths_x_old(j+1,:) &
Shortened_Paths_y_old(j,:)==Shortened_Paths_y_old(j+1,:),
        Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
        Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
        break
    else
       Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
       Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Updating the Costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
szsp_perm=size(Shortened_Paths_x,2);
permcost=zeros(nuav*ntarg,1);

for z=1:szsp_perm,
    [permcost(z,1)]=update_cost([Shortened_Paths_x(:,z),Shortened_Paths_y(:,z)],THREATS);
end
totalcost=reshape(permcost,ntarg,nuav)';
```

shorten_paths.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [shr_x,shr_y]=shorten_paths(sp_x,sp_y,Z,T,spo_x,spo_y)

%INPUTS:
%
%sp - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
%
%Z - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%T - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%spo - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n. This matrix is the original matrix without the voronoi segements
%split up.
%
%OUTPUTS:
%
%shr - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
spo=[spo_x,spo_y];
sp=[sp_x,sp_y];
SC=0;FC=0;SF=0;SN=0;
for j=1:size(T,2),
   PC=[];
   for i=1:size(spo,1)-1,
      SC=sqrt(((T(1,j)-spo(i,1))^2)+((T(2,j)-spo(i,2))^2));
      FC=sqrt(((T(1,j)-spo(i+1,1))^2)+((T(2,j)-spo(i+1,2))^2));
      SF=sqrt(((spo(i+1,1)-spo(i,1))^2)+((spo(i+1,2)-spo(i,2))^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0
         PC(i)=sqrt(SC^2-SN^2);
      else
         if SC<FC
            PC(i)=SC;
         else
            PC(i)=FC;
         end
      end
      mPC=min(PC);
      if  mPC< T(3,j),
         T(3,j)=mPC*.995;
      end
```

```
    end
end

ZT=[Z([1:3],:) T([1:3],:)];
szzt=size(ZT,2);
szsp=size(sp,1);
shr=ones(szsp,2);
for i=1:2,
    shr(:,i)=sp(szsp,i);
end
shr(1,:)=sp(1,:);
a=1;
PC=zeros(1,szzt);
while shr(a,:)~=sp(szsp,:),
    for i=1:szsp,
        if shr(a,:)==sp(i,:)
            pck=i;
            break
        end
    end
    for i=szsp:-1:pck+1,
        SF=sqrt(((shr(a,1)-sp(i,1))^2)+((shr(a,2)-sp(i,2))^2));
        for j=1:szzt,
            SC=sqrt(((ZT(1,j)-shr(a,1))^2)+((ZT(2,j)-shr(a,2))^2));
            FC=sqrt(((ZT(1,j)-sp(i,1))^2)+((ZT(2,j)-sp(i,2))^2));
            SN=(SC^2+SF^2-FC^2)/(2*SF);
            if SN<SF & SN>0
                PC(1,j)=sqrt(SC^2-SN^2);
            else
                if SC<FC
                    PC(1,j)=SC;
                else
                    PC(1,j)=FC;
                end
            end
        end
        if PC(1,:)>ZT(3,:),
            a=a+1;
            shr(a,:)=sp(i,:);
            break
        end
    end
end
shr_x=shr(:,1);
shr_y=shr(:,2);
```

fillet_path.m
Authored by Matthew Lechliter

```
function [Shortened_Paths_fillet_x,Shortened_Paths_fillet_y]=fillet_path(Shortened_Paths,min_turn)

%INPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%min_turn - minimum turning radius for the UAVs

%OUTPUTS:
%
%Shortened_Paths_fillet - is a nxmx2 matrix where n is the length of the
%longest path with the addition of fillets ((2*old size)-1) and m is the
%number of UAVs multiplied by the number of targets.  The element (nxmx1)
%x position of the mth uav at point n.  The element (nxmx2) y position of
%the mth uav at point n.


Shortened_Paths_fillet=Shortened_Paths*0;
Shortened_Paths_fillet(:,1)=Shortened_Paths(size(Shortened_Paths,1),1);
Shortened_Paths_fillet(:,2)=Shortened_Paths(size(Shortened_Paths,1),2);
Shortened_Paths_fillet(1,:)=Shortened_Paths(1,:);

fillet_counter=2;
for j=2:size(Shortened_Paths,1)-1,
   if Shortened_Paths(j,:)==Shortened_Paths(j+1,:),
      break
   end
   start=Shortened_Paths(j-1,:);
   middle=Shortened_Paths(j,:);
   finish=Shortened_Paths(j+1,:);
   SM=sqrt(sum((middle-start).^2));
   MF=sqrt(sum(((finish-middle).^2)));
   SF=sqrt(sum(((finish-start).^2)));
   alpha=acos((SM^2+MF^2-SF^2)/(2*SM*MF));
   Fillet=min_turn/tan(alpha/2);
   if Fillet>=SM
      Shortened_Paths_fillet(fillet_counter,:)=Shortened_Paths(j-1,:);
   else
      Shortened_Paths_fillet(fillet_counter,:)=Shortened_Paths(j-1,:)+(Shortened_Paths(j,:)-
Shortened_Paths(j-1,:))*((SM-Fillet)/SM);
   end
   if Fillet>=MF,
      Shortened_Paths_fillet(fillet_counter+1,:)=Shortened_Paths(j+1,:);
   else
      Shortened_Paths_fillet(fillet_counter+1,:)=Shortened_Paths(j,:)+(Shortened_Paths(j+1,:)-
Shortened_Paths(j,:))*(Fillet/MF);
   end
   fillet_counter=fillet_counter+2;
end
Shortened_Paths_fillet_x=Shortened_Paths_fillet(:,1);
Shortened_Paths_fillet_y=Shortened_Paths_fillet(:,2);
```

heading_angle_paths.m
Authored by Matthew Lechliter

```
function [Shortened_Paths_heading_angle_x,Shortened_Paths_heading_angle_y]=
heading_angle_paths(Shortened_Paths,min_turn,HEADING_ANGLE,num_segs);

warning off MATLAB:divideByZero

if HEADING_ANGLE < 0,
   HEADING_ANGLE=pi*2+HEADING_ANGLE;
end

delta_x = Shortened_Paths(2,1) - Shortened_Paths(1,1);
delta_y = Shortened_Paths(2,2) - Shortened_Paths(1,2);

NEW_HEADING_ANGLE=(atan(abs(delta_y)/abs(delta_x)));
if delta_x>=0 & delta_y>=0,
   NEW_HEADING_ANGLE=NEW_HEADING_ANGLE;
end
if delta_x<0 & delta_y>=0,
   NEW_HEADING_ANGLE=pi-NEW_HEADING_ANGLE;
end
if delta_x<0 & delta_y<0,
   NEW_HEADING_ANGLE=pi+NEW_HEADING_ANGLE;
end
if delta_x>=0 & delta_y<0,
   NEW_HEADING_ANGLE=2*pi-NEW_HEADING_ANGLE;
end

% x and y are the initial positions of the UAV
x=Shortened_Paths(1,1);
y=Shortened_Paths(1,2);

% Rotated heading angle
ROTATED_HEADING_ANGLE=HEADING_ANGLE-NEW_HEADING_ANGLE;

% Rotated NEW_HEADING_ANGLE is 0 degrees
ROTATED_NEW_HEADING_ANGLE=0;

% This section ensures that ROTATED_HEADING_ANGLE is between -pi and pi
if abs(ROTATED_HEADING_ANGLE) > pi
   if ROTATED_HEADING_ANGLE > 0
      ROTATED_HEADING_ANGLE = ROTATED_HEADING_ANGLE-2*pi;
   else
      ROTATED_HEADING_ANGLE = ROTATED_HEADING_ANGLE+2*pi;
   end
end

if abs(ROTATED_HEADING_ANGLE) < pi/5.5
   small_ang=1;
else
   small_ang=0;
   %  Equation found by numerical methods, used to find the location of the
   %  first point to break from the old path onto the first circle

init_dist=0.082565052*(abs(ROTATED_HEADING_ANGLE)/pi*(2*min_turn))^3+0.020254038*(abs(R
```

```
OTATED_HEADING_ANGLE)/pi*(2*min_turn))^2+0.629231718*(abs(ROTATED_HEADING_ANGL
E)/pi*(2*min_turn));

   %  xu and yu are the coordinates of the first point that breaks from the
   %  old path and onto the new path following the circles
   xu = x+init_dist*cos(ROTATED_HEADING_ANGLE);
   yu = y+init_dist*sin(ROTATED_HEADING_ANGLE);


   if ROTATED_HEADING_ANGLE >= 0
      ccw = -1;
   else
      ccw = 1;
   end

   %  Finds the locations of the center of both circles, based on whether
   %  the angle made by the intersection of the old and new heading angles
   %  is positive or negative

   xc1 = (x+min_turn*cos(ROTATED_NEW_HEADING_ANGLE + ccw*.5*pi));
   yc1 = (y+min_turn*sin(ROTATED_NEW_HEADING_ANGLE + ccw*.5*pi));

   xc2 = (xu+min_turn*cos(ROTATED_HEADING_ANGLE - ccw*.5*pi));
   yc2 = (yu+min_turn*sin(ROTATED_HEADING_ANGLE - ccw*.5*pi));

   % dx_c2 and dy_c2 are the delta x and delta y between the position of the
   % center of the first break off point and the center of the first circle
   dx_c2 = xu - xc2;
   dy_c2 = yu - yc2;

   % c2_angle is the angle made by the horizon (x-axis) and the line between
   % the break off point and center of the first circle
   c2_angle=(atan(abs(dy_c2)/abs(dx_c2)));
   if dx_c2>=0 & dy_c2>=0,
      c2_angle=c2_angle;
   end
   if dx_c2<0 & dy_c2>=0,
      c2_angle=pi-c2_angle;
   end
   if dx_c2<0 & dy_c2<0,
      c2_angle=pi+c2_angle;
   end
   if dx_c2>=0 & dy_c2<0,
      c2_angle=2*pi-c2_angle;
   end

   % dx_cc and dy_cc are the delta x and delta y between the position of the
   % center of the final circle and the center of the first circle
   dx_cc = (xc1 - xc2);
   dy_cc = (yc1 - yc2);

   % cc_angle is the angle made by the horizon (x-axis) and the line between
   % the position of the center of the final circle and the center of the first circle
   cc_angle=(atan(abs(dy_cc)/abs(dx_cc)));
   if dx_cc>=0 & dy_cc>=0,
      cc_angle=cc_angle;
```

```
end
if dx_cc<0 & dy_cc>=0,
   cc_angle=pi-cc_angle;
end
if dx_cc<0 & dy_cc<0,
   cc_angle=pi+cc_angle;
end
if dx_cc>=0 & dy_cc<0,
   cc_angle=2*pi-cc_angle;
end

if ccw == 1
   if abs(ROTATED_HEADING_ANGLE)>pi/2
      cc_point = (2*pi-cc_angle);
      c2_point = -(2*pi-c2_angle);
   else
      cc_point = (2*pi-cc_angle);
      c2_point = (c2_angle);
   end
else
   if abs(ROTATED_HEADING_ANGLE)>pi/2
      cc_point = ccw*(cc_angle);
      c2_point = -1*ccw*(c2_angle);
   else
      cc_point = ccw*(cc_angle);
      c2_point = ccw*(2*pi-c2_angle);
   end
end

counter = 1;
for i = (ccw*2*pi/num_segs:ccw*2*pi/num_segs:cc_point+c2_point)+pi/2-c2_angle
   x_c2(1,counter)=min_turn*sin(i)+xc2;
   y_c2(1,counter) = min_turn*cos(i)+yc2;
   counter = counter + 1;
end

dx_c1 = x - xc1;
dy_c1 = y - yc1;

c1_angle=(atan(abs(dy_c1)/abs(dx_c1)));
if dx_c1>=0 & dy_c1>=0,
   c1_angle=c1_angle;
end
if dx_c1<0 & dy_c1>=0,
   c1_angle=pi-c1_angle;
end
if dx_c1<0 & dy_c1<0,
   c1_angle=pi+c1_angle;
end
if dx_c1>=0 & dy_c1<0,
   c1_angle=2*pi-c1_angle;
end

cc_angle=cc_angle+ccw*pi;

counter = 1;
```

```
    for i = (-ccw*2*pi/num_segs:-ccw*2*pi/num_segs:(cc_angle-c1_angle))-(cc_angle-pi/2)
        x_c1(1,counter)=min_turn*sin(i)+xc1;
        y_c1(1,counter) = min_turn*cos(i)+yc1;
        counter = counter + 1;
    end

    %  Rotation back to original coordinates
    [t,r] = cart2pol(xu - x,yu - y);
    t = t + NEW_HEADING_ANGLE;
    [xu_temp,yu_temp] = pol2cart(t,r);

    Shortened_Paths_heading_angle_x_temp(1) = x;
    Shortened_Paths_heading_angle_y_temp(1) = y;
    Shortened_Paths_heading_angle_x_temp(2) = xu_temp + x;
    Shortened_Paths_heading_angle_y_temp(2) = yu_temp + y;

    for i = 1:size(x_c2,2)
        [t,r] = cart2pol(x_c2(i) - x,y_c2(i) - y);
        t = t + NEW_HEADING_ANGLE;
        [x_c2_temp,y_c2_temp] = pol2cart(t,r);
        Shortened_Paths_heading_angle_x_temp(size(Shortened_Paths_heading_angle_x_temp,2)+1) =
(x_c2_temp + x);
        Shortened_Paths_heading_angle_y_temp(size(Shortened_Paths_heading_angle_y_temp,2)+1) =
(y_c2_temp + y);
    end


    for i = 1:size(x_c1,2)
        [t,r] = cart2pol(x_c1(i) - x,y_c1(i) - y);
        t = t + NEW_HEADING_ANGLE;
        [x_c1_temp,y_c1_temp] = pol2cart(t,r);
        Shortened_Paths_heading_angle_x_temp(size(Shortened_Paths_heading_angle_x_temp,2)+1) =
(x_c1_temp + x);
        Shortened_Paths_heading_angle_y_temp(size(Shortened_Paths_heading_angle_y_temp,2)+1) =
(y_c1_temp + y);
    end
end

if small_ang==0,
    sze = size(Shortened_Paths,1);
    Shortened_Paths_heading_angle_x=ones(sze,1)*Shortened_Paths(end,1);
    Shortened_Paths_heading_angle_y=ones(sze,1)*Shortened_Paths(end,2);

    szpts=size(Shortened_Paths_heading_angle_x_temp,2);

    Shortened_Paths_heading_angle_x([1:szpts],1)=Shortened_Paths_heading_angle_x_temp';
    Shortened_Paths_heading_angle_x([szpts+1:sze],1)=Shortened_Paths([1:sze-szpts],1);
    Shortened_Paths_heading_angle_y([1:szpts],1)=Shortened_Paths_heading_angle_y_temp';
    Shortened_Paths_heading_angle_y([szpts+1:sze],1)=Shortened_Paths([1:sze-szpts],2);
else
    Shortened_Paths_heading_angle_x=Shortened_Paths(:,1);
    Shortened_Paths_heading_angle_y=Shortened_Paths(:,2);
end
```

update_cost.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [permcost]=update_cost(Shortened_Paths,THREATS)

%INPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.

%OUTPUTS:
%
%permcost - cost associated with the nth UAV going to the mth TARGET


szsp_num=size(Shortened_Paths,1)-1;
nthrts=size(THREATS,2);
permcost=0;

for i=1:szsp_num,
   start_x=Shortened_Paths(i,1);start_y=Shortened_Paths(i,2);
   finish_x=Shortened_Paths(i+1,1);finish_y=Shortened_Paths(i+1,2);
   SF=sqrt(((finish_x-start_x)^2)+((finish_y-start_y)^2));
   for j=1:nthrts,
      SC=sqrt(((THREATS(1,j)-start_x)^2)+((THREATS(2,j)-finish_y)^2));
      FC=sqrt(((THREATS(1,j)-finish_x)^2)+((THREATS(2,j)-finish_y)^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
      else
         if SC<FC,PC=SC;
         else
            PC=FC;
         end
      end
      if PC < THREATS(3,j),SF=SF+(THREATS(4,j)*100);
      end
   end
   permcost=permcost+SF;
end
```

mmkp_task_allocation.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [Selected_Paths_x,Selected_Paths_y]=
mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Paths_y,nuav)

%INPUTS:
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%nuav - number of UAVs

%OUTPUTS:
%
%Selected_Pos - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%MMKP algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[bestcomb,mincost]=mmkp_new(totalcost);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Taking the results from mmkp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Selected_Paths_x=zeros(size(Shortened_Paths_x,1),nuav);
Selected_Paths_y=zeros(size(Shortened_Paths_x,1),nuav);
for i=1:nuav,
   Selected_Paths_x(:,i)=Shortened_Paths_x(:,(nuav)*(i-1)+bestcomb(1,i));
   Selected_Paths_y(:,i)=Shortened_Paths_y(:,(nuav)*(i-1)+bestcomb(1,i));
End
```

mmkp_new.m

Authored by Matthew Lechliter, Zachary Spritze, and Elena Lucci

```
function [bestcomb,mincost]=mmkp_new(totalcost)

%Inputs:
%
%totalcost - is a nxm matrix where n is the total number of uav's and m is
%the total number of targets or paths. Where the element nxm is the cost
%associated with uav "n" choosing target or path "m".
%
%Outputs:
%
%bestcomb - is a 1xn row with n equal to the number or uav's where each
%element of the row represents which path the uav should select to give the
%optimal solution.
%
%mincost - is a scalar number which is sum of the optimal costs for all
%the uav's paths.
nuav=size(totalcost,1);
mincost=inf;
C_new=perms(1:nuav);
for j=1:size(C_new,1),
   sc=0;
   for i=1:nuav,
      sc=sc+totalcost(i,C_new(j,i));
   end
   if sc < mincost
      bestcomb=C_new(j,:);
      mincost = sc;
   end
end
```

vrt_sim_convert.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(shr_x,shr_y,UAVS,distpast)
%
%INPUTS:
%
%shr - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%
%OUTPUTS:
%
%uav_path_x - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These are the x coordinates of the paths.
%
%uav_path_y - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These are the y coordinates of the paths.
%
%time_uav - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These values correspond to the time at which
%the uavs are at coordinates x and y in uav_path_x and uav_path_y.
%
%altitude_uav - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These values correspond to the altitudes that
%the uavs are at when they are at coordinates x and y in uav_path_x and
%uav_path_y.
%
%Threat_range_vrt - is a 1xn vector where n is the number of threats, where
%the first row is the range of the threats at the altitude where the uavs
%are flying.
%
%Zone_range_vrt - is a 1xn vector where n is the number of zones, where
%the first row is the range of the zones at the altitude where the uavs
%are flying.

nuav=size(shr_x,2);
szshrpth=size(shr_x,1);
shr_x=[[shr_x];[shr_x(szshrpth,:)]];
shr_y=[[shr_y];[shr_y(szshrpth,:)]];
uav_path_x=zeros(nuav,szshrpth+1);
uav_path_y=zeros(nuav,szshrpth+1);
for i=1:nuav,
   for j=1:szshrpth,
      if [shr_x(j+1,i),shr_y(j+1,i)]==[shr_x(j,i),shr_y(j,i)] | j==szshrpth,
         lst_pnt_x=shr_x(j,i);
         nxtlst_pnt_x=shr_x(j-1,i);
         lst_pnt_y=shr_y(j,i);
         nxtlst_pnt_y=shr_y(j-1,i);
         dist_pnts=sqrt(((lst_pnt_x-nxtlst_pnt_x)^2)+((lst_pnt_y-nxtlst_pnt_y)^2));
```

```
            last_x=lst_pnt_x+((lst_pnt_x-nxtlst_pnt_x)*(distpast/dist_pnts));
            last_y=lst_pnt_y+((lst_pnt_y-nxtlst_pnt_y)*(distpast/dist_pnts));
            uav_path_x(i,[j+1:szshrpth+1])=last_x;
            uav_path_y(i,[j+1:szshrpth+1])=last_y;
            uav_path_x(i,j)=shr_x(j,i);
            uav_path_y(i,j)=shr_y(j,i);
            break
        else
            uav_path_x(i,j)=shr_x(j,i);
            uav_path_y(i,j)=shr_y(j,i);
        end
    end
end

%Initializing matrixes
time_uav_temp=zeros(nuav,szshrpth+1);
time_uav=zeros(nuav,szshrpth+1);
altitude_uav=zeros(nuav,szshrpth+1);

%Time matrix
for i=1:nuav,
    for j=1:szshrpth,
        shr_dist(i,j)=sqrt((uav_path_x(i,j)-uav_path_x(i,j+1))^2+(uav_path_y(i,j)-uav_path_y(i,j+1))^2);
        time_uav_temp(i,j+1)=shr_dist(i,j)/UAVS(4,i);
    end
    time_uav(i,[2:szshrpth+1])=sum(time_uav_temp(i,:));
    for j=2:szshrpth+1,
        time_uav(i,j)=time_uav(i,j-1)+time_uav_temp(i,j);
    end
end

time_uav=time_uav*1.01;

%Altitude matrix
for i=1:nuav,
    for j=1:szshrpth+1,
        altitude_uav(i,j)=UAVS(3,i);
    end
end
```

plot_uav.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function
plot_uav(UAVS,TARGETS,ZONES,THREATS,uav_path_x,uav_path_y,n_plots,uavs_existing,targ_existing,threats_existing)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Plotting results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure(n_plots);
hold on;
for i=1:2,
    subplot(1,2,i),
    for i=1:size(UAVS,2)
        if uavs_existing(1,i)==1
            plot(UAVS(1,i),UAVS(2,i),'bd');
            text(UAVS(1,i)+5,UAVS(2,i),{i},'FontSize',12,'Color','b');
            axis([5 200 5 200]);
            hold on;
        end
    end
    for i=1:size(TARGETS,2)
        if targ_existing(1,i)==1
            plot(TARGETS(1,i),TARGETS(2,i),'x','Color',[0,.4,0]);
            text(TARGETS(1,i)+5,TARGETS(2,i),{i},'FontSize',12,'Color',[0,0.4,0]);
            axis([5 200 5 200]);
            hold on;
        end
    end
    for i=1:size(THREATS,2)
        if threats_existing(1,i)==1
            plot(THREATS(1,i),THREATS(2,i),'r*');
            text(THREATS(1,i)+5,THREATS(2,i),{i},'FontSize',12,'Color','r')
            axis([5 200 5 200]);
            hold on;
        end
    end
    hold on;
end

%Plotting Threats and range
for i=1:size(THREATS,2)
    if threats_existing(1,i)==1
        t_threat = (1/32:1/32:1)'*2*pi;
        x_threat = THREATS(3,i)*sin(t_threat)+THREATS(1,i);
        y_threat = THREATS(3,i)*cos(t_threat)+THREATS(2,i);
        for i=1:2,
            subplot(1,2,i),plot(x_threat,y_threat,'r.');hold on;
        end
    end
end

%Plotting No fly Zones
for i=1:size(ZONES,2)
    t_nfz = (1/16:1/16:1)'*2*pi;
    x_nfz = ZONES(3,i)*sin(t_nfz)+ZONES(1,i);
    y_nfz = ZONES(3,i)*cos(t_nfz)+ZONES(2,i);
```

```
    for i=1:2,
        subplot(1,2,i),fill(x_nfz,y_nfz,'k');hold on;
    end
end


%Plotting shortened paths
for i=1:size(uav_path_x,1)
    subplot(1,2,2),plot(uav_path_x(i,:),uav_path_y(i,:),'b-');hold on;
end

subplot(1,2,1),title('Initial Positions');hold on;
subplot(1,2,2),title('Shortened Selected Paths');hold on;
for i=1:2,
    subplot(1,2,i),axis([-25 250 -25 250]);hold on;
end
```

Appendix B

MATLAB Codes for Simulation

place_waypoints_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] =place_waypoints_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*4+9;
sizes.NumInputs=     9*4+9*4;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=place_waypoints(u);

% End of mdlOutputs.
```

place_waypoints.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=place_waypoints(u)

UAVS=u([1:36],1);
UAVS=reshape(UAVS,4,9);


uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end

TARGETS_REAL=u([37:72],1);
TARGETS_REAL=reshape(TARGETS_REAL,4,9);
n_uav=0;n_targ=0;

TARGETS=zeros(4,9);
targets_location=zeros(1,9);
for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      n_uav=n_uav+1;
   end
   if abs(sum(TARGETS_REAL(:,i)))>0
      n_targ=n_targ+1;
   end
end

if n_uav < n_targ
   for i = 1:n_uav
      A=TARGETS_REAL(3,:);
      B=sort(A);
      Column=find(A==B(1,size(B,2)));
      TARGETS(1,i) = TARGETS_REAL(1,Column(1,1));
      TARGETS(2,i) = TARGETS_REAL(2,Column(1,1));
      TARGETS(3,i) = TARGETS_REAL(3,Column(1,1));
      TARGETS(4,i) = TARGETS_REAL(4,Column(1,1));
      targets_location(1,i)=Column(1,1);
      TARGETS_REAL(3,Column(1,1))=0;
   end
else
   counter=1;
   for i=1:9
      if abs(sum(TARGETS_REAL(:,i)))>0
         TARGETS(:,counter)=TARGETS_REAL(:,i);
         targets_location(1,counter)=i;
         counter=counter+1;
      end
   end
end

if n_uav > n_targ
   for i=1:(n_uav-n_targ)
      A=TARGETS_REAL(3,:);
```

```
        B=sort(A);
        Column=find(A==B(1,size(B,2)));
        TARGETS(1,n_targ+i) = i*.01+TARGETS_REAL(1,Column(1,1));
        TARGETS(2,n_targ+i) = i*.01+TARGETS_REAL(2,Column(1,1));
        TARGETS(3,n_targ+i) = 0;
        TARGETS(4,n_targ+i) = 0;
        TARGETS_REAL(3,Column(1,1))=0.5*TARGETS_REAL(3,Column(1,1));
        targets_location(1,i+n_targ)=Column(1,1);
    end
end
TARGETS=[TARGETS,zeros(4,9-size(TARGETS,2))];

sys=[reshape(TARGETS,36,1);targets_location'];
```

path_planning_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] = path_planning_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,
  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization
  case 3
    sys = mdlOutputs(u); % Calculate outputs
  case { 1, 2, 4, 9 }
    sys = [ ]; % Unused flags
  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;
%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*100*4+9;
sizes.NumInputs=     36+36+30+60+1+1+9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = [ ]; % No continuous states
%
str = [ ]; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u)
[sys]=path_planning(u);
% End of mdlOutputs.
```

uav_crash_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] =uav_crash_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9;
sizes.NumInputs=     57;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=uav_crash(u);

% End of mdlOutputs.
```

uav_crash.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=uav_crash(u)

uav_pos=reshape(u([1:27],1),3,9);
zone_pos=reshape(u([28:57],1),3,10);

uav_shot_down=zeros(9,1);

for i=1:9,
   for j=1:10,
      dist_uav_zone=sqrt(((uav_pos(1,i)-zone_pos(1,j))^2)+((uav_pos(2,i)-zone_pos(2,j))^2));
      if dist_uav_zone < zone_pos(3,j),
         uav_shot_down(i,1)=1;
      end
   end
end
sys=[uav_shot_down];
```

uav_intercepted_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] =uav_intercepted_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    24;
sizes.NumInputs=     87;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=uav_intercepted(u);

% End of mdlOutputs.
```

uav_intercepted.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=uav_intercepted(u)

uav_pos=reshape(u([1:27],1),3,9);
threat_pos=reshape(u([28:87],1),4,15);

uav_shot_down=zeros(9,1);
threats_fired=zeros(15,1);
for i=1:9,
   for j=1:15,
      dist_uav_threat=sqrt(((uav_pos(1,i)-threat_pos(1,j))^2)+((uav_pos(2,i)-threat_pos(2,j))^2));
      if dist_uav_threat < threat_pos(3,j),
         threats_fired(j,1)=1;
         uav_chance=rand;
         if uav_chance <= threat_pos(4,j),
            uav_shot_down(i,1)=1;
         end
      end
   end
end
sys=[uav_shot_down; threats_fired];
```

target_classifier_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
 function [sys,x0,str,ts] = target_classifier_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    36;
sizes.NumInputs=     100;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=target_classifier(u);


% End of mdlOutputs.
```

target_classifier.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=target_classifier(u)

TARGETS_OLD=u([1:36],1);
TARGETS_OLD=reshape(TARGETS_OLD,4,9);

END_OF_PATH=u([37:45],1);

SELECTED_TARGETS=u([46:54],1);

TARGETS_REAL=u([55:90],1);
TARGETS_REAL=reshape(TARGETS_REAL,4,9);

target_location=u([91:99],1);

clock=round(u(100,1));

uav_complete=find(END_OF_PATH==1);
nuav_complete=size(uav_complete,2);
action=0;
for i=1:nuav_complete,
   target_real_location=target_location(SELECTED_TARGETS(uav_complete(1,i),1));
   action=TARGETS_REAL(4,target_real_location);
   if TARGETS_REAL(4,target_real_location) < 4,
      TARGETS_REAL(4,target_real_location)=TARGETS_REAL(4,target_real_location)+1;
   else
      TARGETS_REAL(:,target_real_location)=0;
   end
   if action==1,
      target_present=rand;
      if target_present <= .9,
         disp(sprintf('Target %d (value %d) indentified as a target at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
      else
         disp(sprintf('Target %d (value %d) indentified as NOT a target at time %d by UAV %d.',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
         disp(sprintf('Target %d has been removed from target status at time %d.\n',...
         target_real_location,clock));
         TARGETS_REAL(:,target_real_location)=0;
      end
   end
   if action==2, disp(sprintf('Target %d (value %d) classified not attacked at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i))); end
   if action==3, disp(sprintf('Target %d (value %d) attacked not assested at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i))); end
   if action==4,
      target_destroyed=rand;
      if target_destroyed <= .85,
         disp(sprintf('Target %d (value %d) assested as destroyed at time %d by UAV %d. \n',...
            target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
      else
         disp(sprintf('Target %d (value %d) assested as NOT destroyed at time %d by UAV %d. \n',...
            target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
         TARGETS_REAL(4,target_real_location)=3;
      end
```

```
    end
end

if sum(sum(TARGETS_REAL))==0,
    TARGETS_REAL(:,1)=[4 2 3 1]';
end


sys=reshape(TARGETS_REAL,36,1);
```

compare_targets_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] =compare_targets_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    36;
sizes.NumInputs=     36*9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = [ ]; % No continuous states
%
str = [ ]; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=compare_targets(u);

% End of mdlOutputs.
```

compare_targets.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=compare_targets(u)

targets_1=reshape(u([1:36],1),4,9);
targets_2=reshape(u([37:72],1),4,9);
targets_3=reshape(u([73:108],1),4,9);
targets_4=reshape(u([109:144],1),4,9);
targets_5=reshape(u([145:180],1),4,9);
targets_6=reshape(u([181:216],1),4,9);
targets_7=reshape(u([217:252],1),4,9);
targets_8=reshape(u([253:288],1),4,9);
targets_9=reshape(u([289:324],1),4,9);

for i = 1:9
   real_targets(:,i) = targets_1(:,i);
   if targets_2(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_2(:,i);
   end
   if targets_3(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_3(:,i);
   end
   if targets_4(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_4(:,i);
   end
   if targets_5(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_5(:,i);
   end
   if targets_6(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_6(:,i);
   end
   if targets_7(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_7(:,i);
   end
   if targets_8(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_8(:,i);
   end
   if targets_9(4,i)>real_targets(4,i)
      real_targets(:,i) = targets_9(:,i);
   end
end

sys=reshape(real_targets,36,1);
```

compare_threats_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] =compare_threats_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    60;
sizes.NumInputs=     60*9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = [ ]; % No continuous states
%
str = [ ]; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=compare_threats(u);

% End of mdlOutputs.
```

compare_threats.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys]=compare_threats(u)

threats_1=reshape(u([1:60],1),4,15);
threats_2=reshape(u([61:120],1),4,15);
threats_3=reshape(u([121:180],1),4,15);
threats_4=reshape(u([181:240],1),4,15);
threats_5=reshape(u([241:300],1),4,15);
threats_6=reshape(u([301:360],1),4,15);
threats_7=reshape(u([361:420],1),4,15);
threats_8=reshape(u([421:480],1),4,15);
threats_9=reshape(u([481:540],1),4,15);
for i = 1:15
   real_threats(:,i) = threats_1(:,i);
   if threats_2(4,i) == 0
      real_threats(:,i) = threats_2(:,i);
   end
   if threats_3(4,i) == 0
      real_threats(:,i) = threats_3(:,i);
   end
   if threats_4(4,i) == 0
      real_threats(:,i) = threats_4(:,i);
   end
   if threats_5(4,i) == 0
      real_threats(:,i) = threats_5(:,i);
   end
   if threats_6(4,i) == 0
      real_threats(:,i) = threats_6(:,i);
   end
   if threats_7(4,i) == 0
      real_threats(:,i) = threats_7(:,i);
   end
   if threats_8(4,i) == 0
      real_threats(:,i) = threats_8(:,i);
   end
   if threats_9(4,i) == 0
      real_threats(:,i) = threats_9(:,i);
   end
end

sys=reshape(real_threats,60,1);
```

display_initial_s.m

Authored by Matthew Lechliter and Zachary Spritzer

```
function [sys,x0,str,ts] = display_initial_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    0;
sizes.NumInputs=     36+36+30+60;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = [ ]; % No continuous states
%
str = [ ]; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function mdlOutputs(u)


display_initial(u);

% End of mdlOutputs.
```

display_initial.m
Authored by Matthew Lechliter and Zachary Spritzer

```
function display_initial(u)

UAVS=u([1:4*9],1);
UAVS=reshape(UAVS,4,9);
a=4*9;
TARGETS=u([a+1:a+4*9]);
TARGETS=reshape(TARGETS,4,9);
a=a+4*9;
ZONES=u([a+1:a+3*10]);
ZONES=reshape(ZONES,3,10);
a=a+3*10;
THREATS=u([a+1:a+4*15]);
THREATS=reshape(THREATS,4,15);

for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      disp(sprintf('UAV %d exists at location %d x, location %d y, altitude %d km, and is flying at %d m/s.
\n',...
         i,round(UAVS(1,i)),round(UAVS(2,i)),round(UAVS(3,i)),round(UAVS(4,i)*1000)));
  end
end

for i=1:9
   if abs(sum(TARGETS(:,i)))>0
      disp(sprintf('Target %d indicated to be at location %d x, location %d y , and with an estimated value
of %d. \n',...
         i,round(TARGETS(1,i)),round(TARGETS(2,i)),round(TARGETS(3,i))));
  end
end

for i=1:10
   if abs(sum(ZONES(:,i)))>0
      disp(sprintf('No-Fly Zone %d exists at location %d x, location %d y, and with a radius of %d km.
\n',...
         i,round(ZONES(1,i)),round(ZONES(2,i)),round(ZONES(3,i))));
  end
end

for i=1:15
   if abs(sum(THREATS(:,i)))>0
      disp(sprintf('Threat %d exists at location %d x, location %d y, with a range of %d km, and has a
probability of kill of %d%%. \n',...

i,round(THREATS(1,i)),round(THREATS(2,i)),round(THREATS(3,i)),round(THREATS(4,i)*100)));
  end
end
```